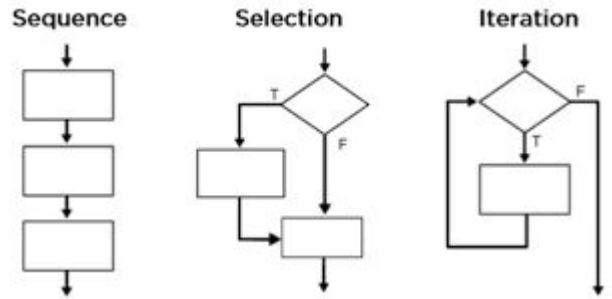
Controlla-palooza

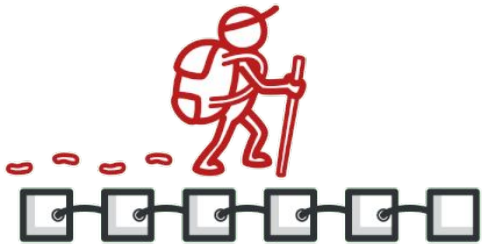
Everything you never learned about how code executes!

Expression Evaluation

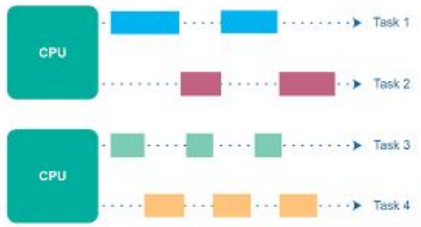
Short Circuiting



Flow Control



Iterators



Asynchronous Programming

Expression Evaluation

What's the big picture?

Expressions like

f(x) + g(x)

are evaluated differently by different languages!

For example, does the language evaluate from right to left, left to right, or is there no specified order?

If these function calls don't have side effects, then the order of execution doesn't matter.

But if these functions do have side effects, the order of evaluation of an expression must be considered!

Expressions and Their Evaluation

Expressions

An expression is a combination of values, function calls and operations that, when evaluated, produces an output value.

An expression may or may not have side effects.

Which of the following are expressions?

```
f(x)*5
```

```
if (foo)
   cout << "bar";
```

```
b = a = 5
```

```
a && (b || c)
```

```
if a > 5 then a*3 else a*4
```

Expressions, Associativity and Order of Evaluation

So if f() has a side effect, this could impact the result!

Challenge: What will these C++ programs p

Similarly if f() has a side effect, the results can change!

```
int c = 1, d = 2;

int f() {
  ++c;
  return 0;
}

int main() {
  cout << c - d + f() ;
}
```

Why? C++ doesn't specify what order the terms of an arithmetic expression must be evaluated!

Nor does it specify what order parameters must be evaluated!

Some languages mandate a left-to-right eval order (C#, Java, JavaScript, Swift), while others don't (Rust, OCaml).

```
int c = 1;

int f() {
  ++c;
  return c;
}

void h(int a, int b)
  { cout << a << " "<< b; }

int main() {
  h(c, f());
}
```

The value of c could be evaluated first…

Either 1 or 0

Or perhaps the value of f() could be evaluated first!

The value of c could be evaluated first!

Either 0 or 1

Or perhaps the value of f() could be evaluated first!

Challenge: What's the rationale behind not specifying a strict order?

Expressions, Associativity

Academic Robot Says:

"There are a few exceptions, Carey:

1. Exponentiation is often right-associative: 2^3^2 → 512
2. Assignment is right associative: a = b = c = 5 → c=5; b=c; a=b;
3. Unary operators are right associative: ~!a → ~ (!a)"

```
int c = 1, d = 2;

int f() {
  ++c;
  return 0;
}

int main() {
  cout <<(c -(d)+ f());
}
```

And not this…

So expressions like this will be evaluated implicitly like this…

Regardless what o… terms are evaluated…

…anguages use left-to-right associativity when evaluating mathematical expressions.

Which is the way we all learned to do it in school.

```
int f() {
  ++c;
  return c;
}

void h(int a, int b)
  { cout << a << " "<< b; }

int main() {
  h(c, f());
}
```

Short Circuiting

Challenge: What will these programs print?

Why? Let's see!

```
int f()
  { cout << "UCLA "; return 5; }

int g()
  { cout << "students "; return 10; }

int h()
  { cout << "are "; return 20; }

int main() {
  if (f() < 10 && g() < 5 && h() > 9)
   cout << "awesome";
}
```

Answer:

UCLA students

```
int f()
  { cout << "CS131 "; return 5; }

int g()
  { cout << "has "; return 10; }

int h()
  { cout << "no "; return 20; }

int main() {
  if (f() < 10 || g() > 5 && h() > 9)
   cout << "rules";
}
```

Answer

CS131 rules

Short Circuiting: How it Works

Consider the expression below:

```
bool cute = true;

if (cute == true || smart == true || rich == true)
    cout << “Hey, want to go on a date?”;
```

If you’re either cute, *or* smart, *or* rich, then I’d like to go out with you.

Let’s say that I happen to know you’re cute.

Once I know you’re cute, I no longer need to waste time asking the other two questions…

Short Circuiting: H

Short circuiting applies to any boolean expression, not just if statements, e.g.,

bool result = cute || smart || rich;

```
bool cute = false; smart = true;

if (cute == true || smart == true || rich == true)
    cout << "Hey, want to go on a date?";
```

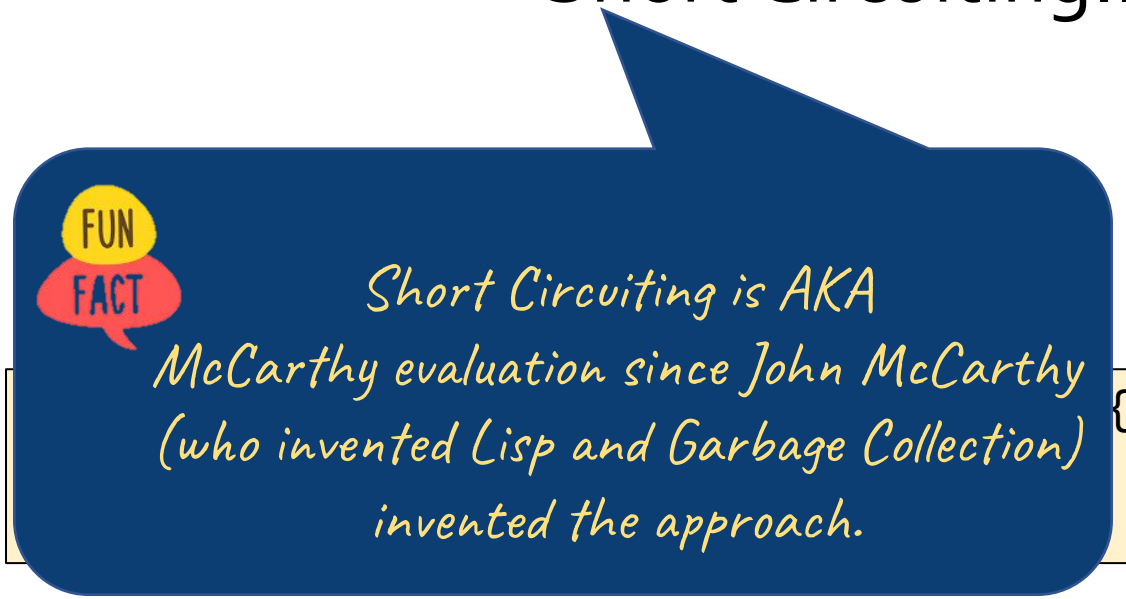
This is called short-circuiting.

In any boolean expression with AND / OR, most languages will evaluate its sub-expressions from left-to-right…

The moment the language finds a condition that satisfies or invalidates the boolean expression, it will skip evaluating the rest of the sub-expressions.

This can dramatically speed up your code!

Short Circuiting… What's Happening?




```
{
```

What's Happening

```
if (is_cute())
  date_person();
else if (is_smart())
  date_person();
else if (is_rich())
  date_person();
```

```
if (good_salary() && fun_work() && good_culture()) {
  take_job();
}
```

```
if (good_salary()) {
  if (fun_work()) {
    if (good_culture()) {
      take_job();
    }
  }
}
```

Short Circuiting

If the user passed in a nullptr, then we abort evaluating the later part of the expression.

Here's a typical use of short-circuiting:

```
void assign_letter_grade(CourseResults *ptr) {
  if (ptr != nullptr && ptr->score > 95)
      ptr->grade = "A+";
  ...
}
```

This code would crash!

And just continue following co

```
// without short circuiting
void assign_letter_grade(CourseResults *ptr) {
  if (ptr != nullptr) {
     if (ptr->score > 95)
        ptr->grade = "A+";
  }
  ...
}
```

We'd have to introduce a second if statement, perhaps with additional nesting of blocks. Yuck!

This can actually make our code much simpler/less wordy!

Classify That Language: Short Circuiting

```
fun rich(salary: Int): Boolean {
  if (salary > 200000) return true;
  print("Not in silicon valley!\n")
  return false
}

fun main() {
  val cute = true; val smart = false;
  val salary = 200000;

  if (cute == true || smart == true || rich(salary) == true)
    print("You're my type!\n")

  if (cute == true or smart == true or rich(salary) == true)
    print("I still want to date you!\n")
}
```

The code to the left prints out the following:

```
You're my type!
Not in silicon valley!
I still want to date you!
```

What can we say about how this language implements short circuiting?

This is Kotlin!

Control Statements

Control Flow

What's the big picture?

There are three major categories of control flow:

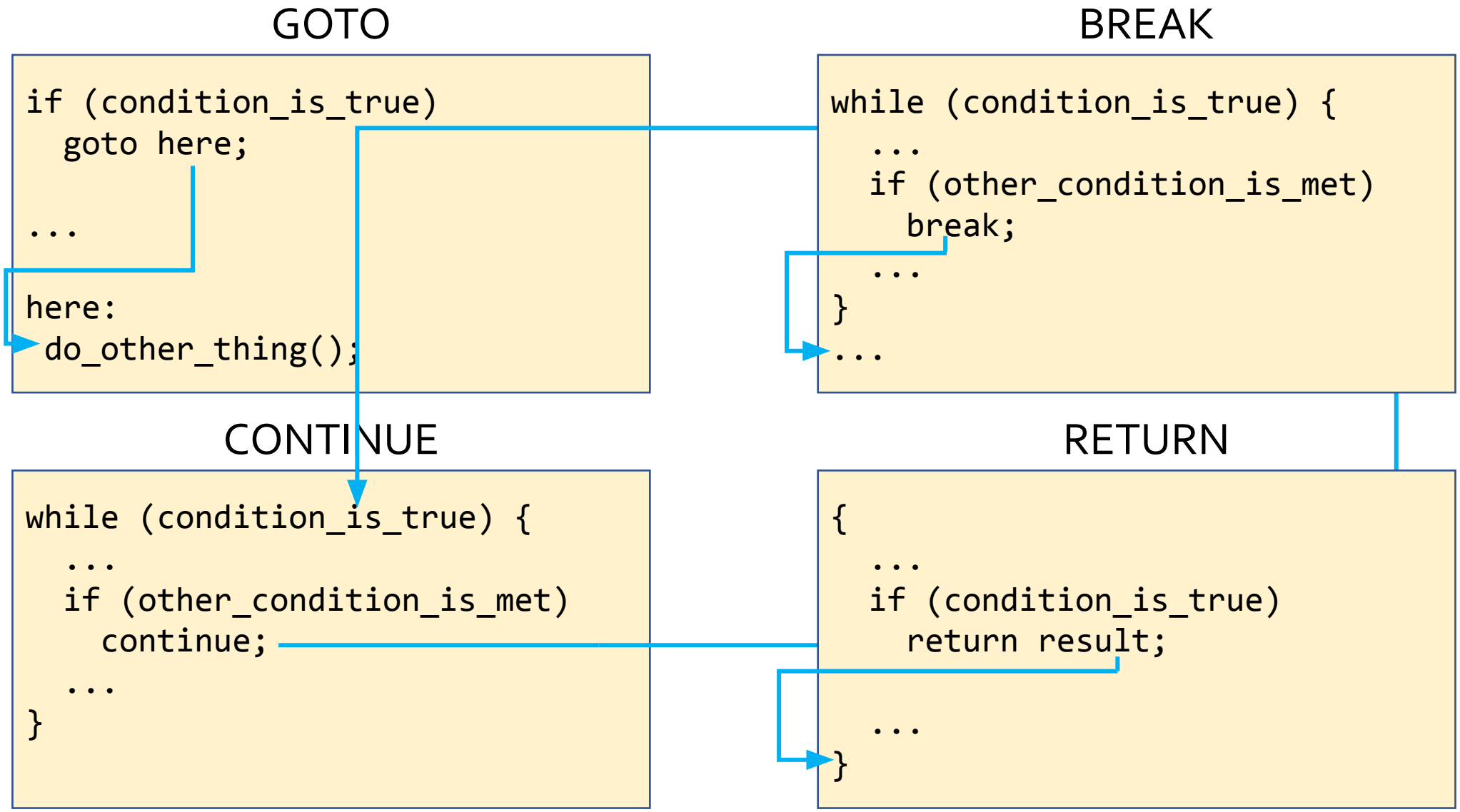
Unconditional Branching Statements:
goto, break, return, and continue

Conditional Statements:
if/else, switch, guards, etc.

Loops:
while, do-while, for, foreach, etc.

Next, we'll do a quick survey of these control structures, and learn some interesting variations.

Unconditional Branching Statements



Let's dive a bit into goto, break and continue.

Unconditional Branching: GOTO

```
// C++ goto example using
// string labels
int main() {
  for (int i=0;i<10;++i) {
    ...
    if (foo(i)) goto done;
    ...
  }
done:
  ...
}
```

A label might be a string…

```
0  REM GOTO EXAMPLE IN BASIC
1  REM USING NUMERIC LABELS
10 PRINT "HELLO WORLD"
20 GOTO 1
```

Or it might be numeric

One of the earliest flow-control statements was GOTO.

In early programming languages, you could assign names (aka labels) to lines in your program.

And then use the GOTO statement to transfer control to a specific line based on its label.

Unconditio

```
// C++ goto example
int main() {
  int i, j;
  cin >> i;
  if (i > 10)
    goto inside;

  for(j=0;j<20;++j) {
    ...
inside:
    cout << j << endl;
  }
}
```

FUN FACT

The Linux Kernel written in C contains roughly 100,000 GOTO statements!

wned

Why?

EWD215 - 0

A Case against the GO TO Statement.

by Edsger W.Dijkstra
Technological University
Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

Unconditional Branching: Break and Continue

In CS31 you learned about the break and continue statements.

So we won't rehash the details…

But, I bet you didn't know that you can use break and continue with labels in some key languages!

Let's see!

Unc

Academic Robot Says:

"A growing number of languages including Java, JavaScript, Kotlin and PHP now have labelled breaks and continues!"

The language finds the loop associated with the label.

```
// JavaScr
outer_loop:
for (i = 0; i < 100000; i++) {
  console.log("outer: " + i)
  for (j = 0; j < 100000; j++) {
    console.log("inner: " + j)
    if (j == 2) break outer_loop;
  }
}
console.log("done!"
```

```
outer: 0
inner: 0
inner: 1
inner: 2
```

The language finds the loop associated with the label.

d break...

Challenge: What output does this

?

```
Kotlin
r@ for (i in 1..3) {
                .00) {
                ")
                @outer
}
println("done!")
```

```
uter: 2
ner: 1
r: 3
r: 1
!
```

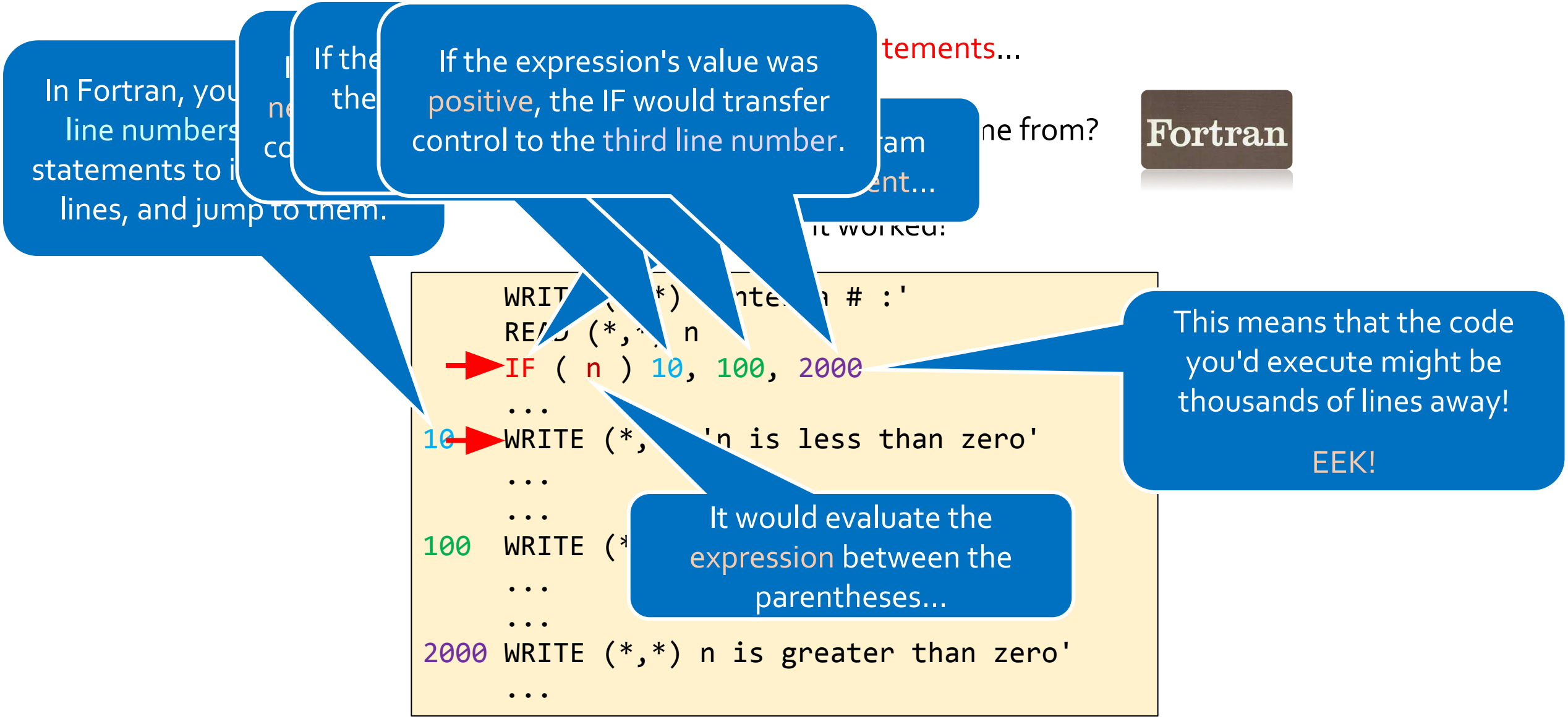
And then exits every nested loop and continues running the line after the end of the labeled loop.

And then immediately breaks out of all inner loops and jumps to the labelled loop for the next iteration.

If you hit a labelled continue...

Conditionals

Conditionals: If Statements



Since if statements are uniform across languages, let's learn some related conditionals.

Conditionals: Beyond If Statements

Hopefully you've seen C++'s conditiona

Here are its semantics: If bo

```
// C++, C#, Java, JavaScript, Swift, et
beverage = age < 21 ? "Coke"  : "Beer";
```

FUN FACT

In Kotlin, this is called the elvis operator because if you turn it on its side, it looks like Elvis!

?:

```
// Kotlin condi
beverage = if age                oke"  else "Beer"
```

```
// Kotlin
fun eat_startup_meal(food: String?) {
  val meal = (food ?: "ramen") + " and redbull"
  println("Time for dinner, let's have $meal.")
}

fun main(args: Array<String>) {
  // Running out of funding
  eat_startup_meal(null)
  // Just raised a round
  eat_startup_meal("sushi")
}
```

```
Time for dinner, let's hav
Time for dinner, let's hav
```

Challenge: Here's a related operator. What does it do?

This is called the null coalescing operator.

A ?: B means:

If A is not NULL return A, else return B

```
// C#, JavaScript, Swift, etc.
meal = food ?? "ramen";
```

Condition

An

Challenge:
output on th

```
if (perks == null)
  return null
else if (perks.weekend_meals == null)
  return null
else
  return perks.weekend_meals.breakfast
```

```
// Javascript
function sunday_brunch_at_the_office(perks) {
  var meal = perks?.weekend_meals?.breakfast
  if (meal != null) console.log("Let's eat " + meal)
               else console.log("You're on your own.")
}

sunday_brunch_at_the_office(null)

var menu = {breakfast: "eggs", lunch: "ramen", dinner: "ribs"}
var perks = {weekday_meals: menu}
sunday_brunch_at_the_office(perks)

perks["weekend_meals"] = menu
sunday_brunch_at_the_office(perks)
```

```
You're on your own.
You're on your own.
Let's eat eggs
```

This is called the
safe navigation operator.

Conditionals: Multi-way Selection

Everyone loves the switch statement...

```
cout << "Enter a number b
cin >> n;
switch (n) {
  case 1:
    cout << "Running code for case 1';
    break;
  case 2:
    cout << "Running coc
    break;
  case 3:
    cout << "Running coc
    break;
  case 4:
    cout << "Running code
    break;
}
```

This is Fortran's equivalent of a switch statement!

Since there are 4 line numbers here...

Fortran expects this value to be between 1-4

Le e what its ancestor looked
Yo essed it – it's from Fortra

```
      WRITE (*,*) 'Enter a # betwe n 1-4:'
      READ (*,*) n
      GO TO ( 10, 25, 50, 100 ), n
      ...
10    WRITE (*,*) 'Running code fo  case 1'
      ...
25    WRITE (*,*) 'Running code f  ase 2'
      ...
50    WRITE (*,*) 'Running code f  se 3'
      ...
100   WRITE (*,*)
      ...
```

If n is out of range, then execution continues here!

Then it jumps to that line to run the instructions there.

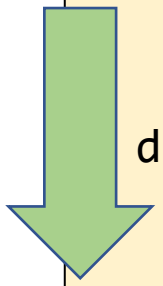
When the code reaches this line, this variable is used to select one of these line numbers.

It was janky, but it worked...

Conditionals: Multi-way Selection

Ok, let's briefly review switch's C++ syntax/semantics, then see variations from different languages.

```
// C++ switch statement

int drinks = 4;

switch (drinks)
{
  case 1:
  case 2:
  case 3:
    cout << "I'm chillin\n";
    break;
  case 4:
    cout << "I'm lit!\n";
    cout << "Life is good!\n";
  default:
    cout << "puke!\n";
    break;
}
```



We can have multiple cases together, but they must be listed one per line.

No { } are required for

All cases must be constant char/int/enum values.

If you don't include a break statement, the program keeps running until you hit one.

Conditionals: Multi-way Selection

```
// C++ switch statement

int drinks = 4;

switch (drinks)
{
  case 1:
  case 2:
  case 3:
    cout << "I'm
    break;
  case 4:
    cout << "I'm lit!\n";
    cout << "Life is good!\n";
  default:
    cout <
    break;
}
```

You can have non-scalar values for your labels.

You don't need break to prevent fall-through.

Oh, and BTW, Swift's switch must be exhaustive, or have a default case for it to compile!

Challenge: This Swift code produces the following output. How does Swift's switch differ from C++'s?

```
// Swift
var name = "Paul"

switch name {
  case "Carey":
    print("Haskell is easiest")
    print("Smalltalk is confusing")
  case "Paul", "Todd":
    print("OCaml is lit!")
  default:
    print("Fortran rocks!")
}
```

Ocaml is lit!

Conditionals: Multi-way Selection

Challenge: What are all the ways Kotlin's switch statement differs from C++'s?

```
// C++ switch statement

int drinks = 4;

switch (drinks)
{
  case 1:
  case 2:
  cas
    c
    b
  cas
    cout << "I'm lit!\n";
    cout << "Life is good!\n";
  default:
    cout << "puke!\n";
    break;
}
```

You can use ranges for labels.

Break is not m… at the end… does…

We use "else" instead of default.

```
// Kotlin
var drinks: Int = 4;

when (drinks) {
  in 1..3 -> println("I'm chillin")
  4 -> {
    println("I'm lit!")
    println("Life is good!")
  }
  else -> println("puke!");
}
```

We need bracers to have multiple statements.

```
I'm lit
Life is good!
```

Conditionals: Multi-way Selection

Challenge: What are all the ways Python's switch differs from C++'s?

```
// C++ switch statement

int dri

switch
{
  case 1:
  case 2:
  case 3:
    cout <
    break;
  case 4:
    cout << "I'm lit!\n";
    cout << "Life is good!\n";
  default:
    cout << "puke!\n";
    break;
}
```

You can match against strings, lists, and dictionaries.

There's no break statement needed to prevent fall-thru.

```
def greet(x):
  match x:
    case 'person':
      print('Hi person!')
    case [p1, p2]:
      print(f'Hi {p1} and {p2}!')
    case {'name': n, 'age': a}:
      print(f"Welcome {n}, you're {a}")
    case p if 'berg' not in p:
      print(f'Hello, {p}!')
    case p:
      print(f'Go        {p}!')

greet('Bjarne
```

You can pattern match and refer to variables from the pattern!

You can place conditions on your matches.

```
Hello, Bjarne!
```

Conditionals: Multi-way Selection

```
// C++ switch statement

int dr

switch
{
  case 1:
  case 2:
  case 3:
    cout << "I'm chillin\n";
    break;
  case 4:
    cout << "I'm lit!\n";
    cout << "Life is good!\n";
  default:
    cout << "puke!\n";
    break;
}
```

The cases are each boolean expressions – the first one that matches runs.

Challenge: Go has traditional switch statements, and also this form. How does this version differ from C++'s?

```
var drinks int = 1
var dessert = false

switch {
  case drinks < 3:
    print("I'm chillin\n")
  case drinks < 5 && !dessert:
    print("I'm lit!\n")
  default:
    print("puke!\n")
}
```

```
I'm chillin
```

This one's more like guards from functional languages!

Classify That Language: Conditionals

```
func classify(eyes int) {
  switch eyes {
  case 1:
    fmt.Println("Copepod")
  case 8:
    fmt.Println("Spider")
  case 3:
    fmt.Println("Tuatara")
  case 0:
    fmt.Println("Star-nosed Mole")
    fallthrough
  default:
    fmt.Println("Freak!")
  }
}

func main() {
  classify(1)
  classify(0)
}
```

What do you think the program on the left prints out?

```
Copepod
Star-nosed Mole
Freak!
```

In this language, each case breaks automatically when reaching the next case statement!

You must use *fallthrough* to allow one case to fall through to the next.

This is (also) Go!

Iteration (aka Looping)

Looping/Iteration

What's the big picture?

Loops come in three major forms:

Counter-controlled	Condition-controlled	Collection-controlled
for i in range(1,10): do_something()	while (condition) do_something()	for (x in container) do_something(x)

Both collection-controlled (and often) counter-controlled loops rely upon iterators under the hood.

Let's see!

Hist

How did the first hig

FUN FACT

Ada Lovelace, 1815-1852, was the first person to describe the concept of looping - for Babbage's Analytical Engine.

Answer: Fortran used IF and GOTO statements to loop!

```
        N = 10
        I = 0
100     P = N - I
        IF  (P),  200, 200, 101
101     ... statements to run in loop
        I = I + 1
        GO TO 100
200     ... statements after loop
        ... statements after loop
```

Let ... Contr... languag...

Perl provides an anonymous name of _ for the iteration value.

This range notation... an object that has a... called for_eac...

We specify a lambda function with a single argument to for_each()!

In Ruby, numbers appear to be objects, and have an upto(n) method!

Some languages have an inclusive range [A,B] and others do not [A,B).

```
# Perl
for (0..9) {
  print("$_ \n");
}
```

```
// Rust
for elem in 0..10 {
    println!("{}", elem);
}
// Or this:
(0..10).for_each( |elem| println!("{}", elem) );
```

```
// Kotlin
for (elem in 0..9)
  { println("$elem") }
// Or this:
(0..9).forEach(
  { elem -> println("$elem") })
```

```
# Ruby
(0..9).each do |elem|
  print elem, " "
end
# Or this:
0.upto(9)
  { |elem| print elem, " " }
```

```
for elem in range(0,10):
    print(elem)
```

```
// PHP
foreach (range(0,9) as $i) {
  echo $i, " ";
}
```

```
// Go
for i := 0; i < 10; i++ {
  fmt.Println(i)
}
```

```
// Swift
for elem in 0...9 {
  print(elem)
}
// Or this:
(0...9).forEach
  { elem in print(elem) }
```

What interesting things do you notice about these approaches?

Collection-c

Let's see how a variety of languages [illegible] over items in a collection!

Many languages have similar to C++'s:

for (type elem: items) { … }

```
# Perl
@items = ("fee","fi","fo","fum");

foreach $elem (@items)
  { print "$elem\n"; }
```

```
# Ruby
items = ["fee","fi","fo","fum"]

items.each do |elem|
    print(elem,"\n")
end
```

```
// Rust
let ite[illegible] = vec!["fe

for elem in items
  { println!("{}",elem

// Or this:
items.iter().for_each(|elem| println!("{}", elem));
```

The other approach is to use a collection.for_each({run code})

We provide for_each() with a lambda or block of code.

```
// Kotlin
var items = arrayOf("fee","fi","fo[illegible]m")
for (elem in items) println(elem)

// Or this:
items.forEach({ elem -> println("$elem") })
```

```
// Swift
et items = ["fee","fi","fo","fum"]

or elem in items { print(elem); }
// Or this:
items.forEach { elem in print(elem) }
```

```
// PHP
$items = array("fee","fi","fo","fum");

foreach ($items as $elem)
  { echo("$elem\n"); }
```

What interesting things do you notice about these approaches?

What about While/Do-While Loops?

OK... May t Smalltalk!

```
" Smalltalk while loop "
number := 10.
[ number > 0 ] whileTrue:
  [ Transcript print: number; nl.
    number := number - 1 ]
```

Then we call Boolean's whileTrue method.

This evaluates to a Boolean...

And pass in a lambda (with no arguments) to run.

Alright, enough with basic loops – let's discuss how iteration actually works!

How Does Iteration Actually Work?

We just saw how languages let us use simple looping syntax to iterate through iterable objects like lists, vectors and ranges.

So how does this iteration actually work under the hood?

These kinds of loops are implemented using "iterators"

```
for n in student_names:
 print(n)
```

```
for i in range(0,10):
  print(i)
```

Loops like these are built using first-class functions

```
fruits.forEach
 { f -> println("$f") }
```

Let's dive into each of these approaches.

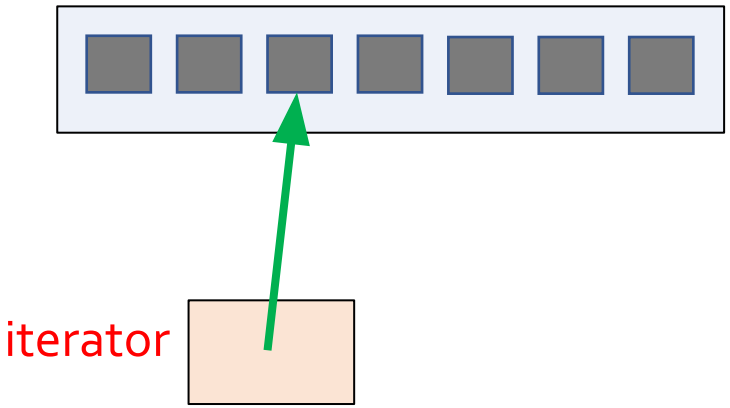
Iterators

An iterator is an object that enables enumeration of a sequence of values.

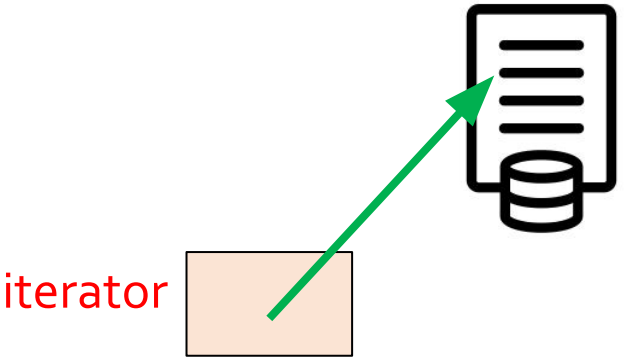
An iterator provides a way to access the values of a sequence without exposing the sequence's underlying implementa

An abstract sequence is one where the values in the sequence aren't stored anywhere, but generated as they're retrieved via the iterator

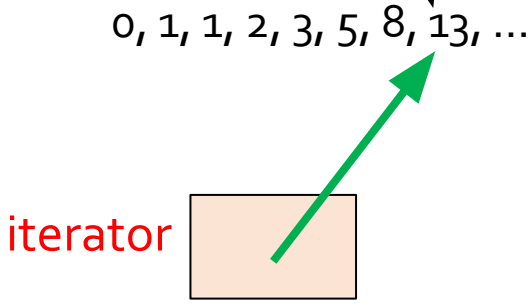
Iterators can be used to enumerate the values held in a container (e.g., a list, set)



Iterators can be used to enumerate the contents of external sources like data files



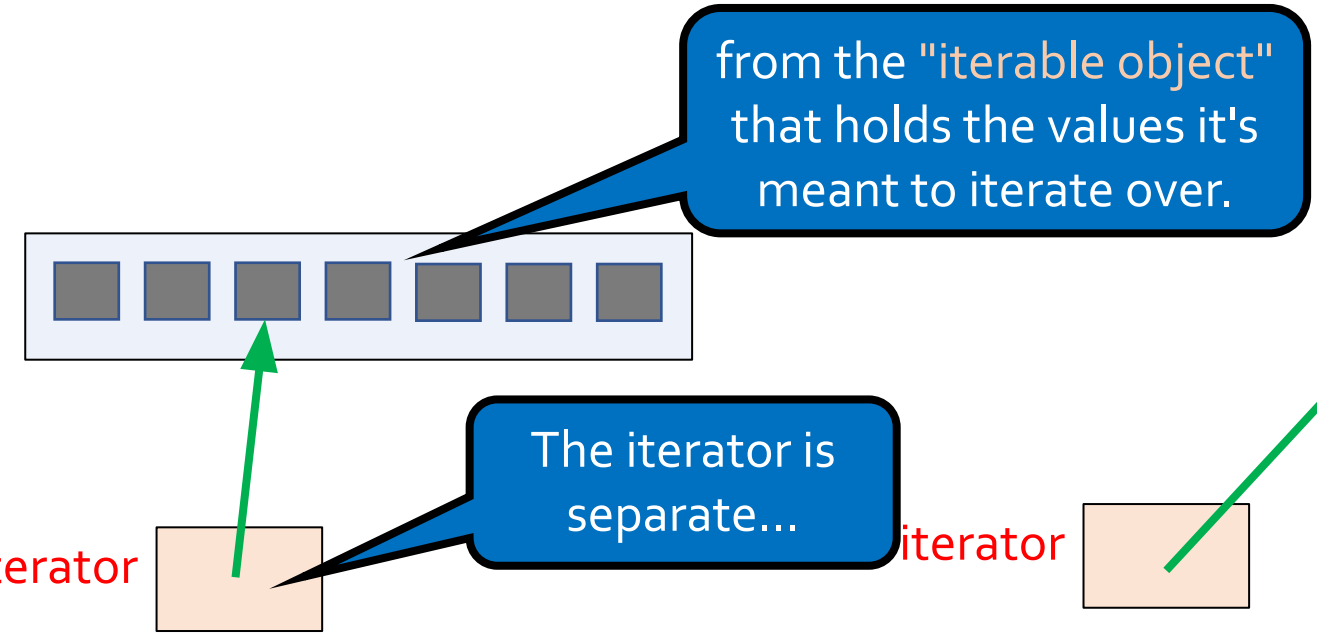
Iterators ca d to enumerate the of an abstract seq e

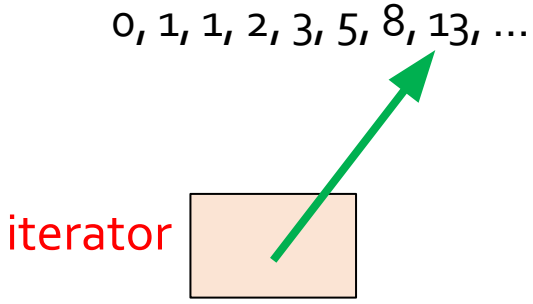


Iterators

Iterators used with containers and external sources are typically distinct objects from the sequences they refer to.

Why do we separate the iterator from the iterable object that actually holds the sequence?





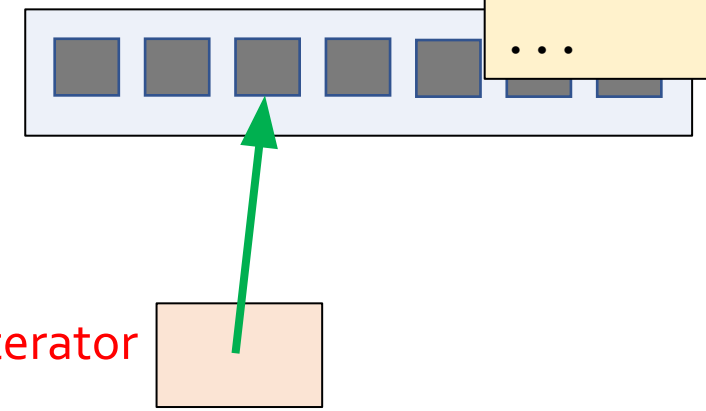
Iterators

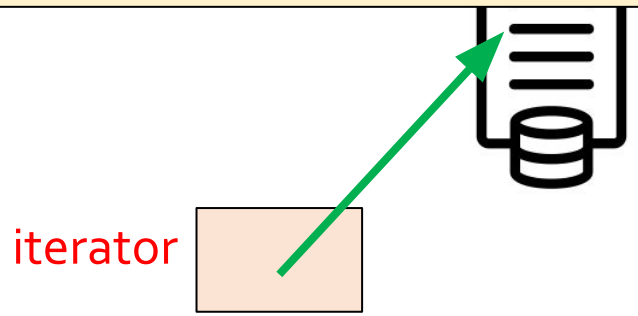
To create an iterator, you must ask the "iterable object" that holds/defines your sequence to give one to you.

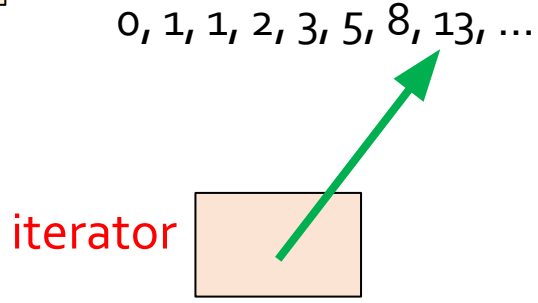
```
// Kotlin: iterator into container
val numbers = listOf(10,20,30,42)

val it = numbers.iterator()
...
```

```
// Swift: abstract sequence
var range = 1...1000000

var it = range.makeIterator()
...
```

```
// C++
vector<string> fruits = {"apple", "pear", ... };

auto it = fruits.begin();
...
```







Iterators

* C++ being a notable exception ☺️

While iterators differ by language, most* tend to have an interface that look like one of these:

For Containers and External Sources

`iter.hasNext()`:
Are there more value(s) pointed to by his iterator (or after it)?

`iter.next()`:
Get the value pointed to by the iterator and advance th

For Abstract Sequences

`iter.next()`:
Generates and returns the next value in the sequence,

OR indicates the sequence is over via a return code/exception

```
// C++ has unusual iterator syntax!
vector<string> fruits = {"apple", "pear", ... };

for (auto it = fruits.begin(); it != fruits.end(); ++it)
  cout << *it;
```

```
// Kotlin: i
val numbers

val it = numbers.iterator()
while (it.hasNext()) {
  val v = it.next();
  println(v)
}
```

ite

```
sequence
00000

var it = range.makeIterator()
while true {
  var v = it.next()
  if (v == nil) { break }
  print(v)
}
```

Iterat

When you loop over the items in a
secretly uses an iterator to r

Academic Robot Says:

"This is a fantastic example of Syntactic Sugar!"

```
// Kotlin: iterate over list of #s
val numbers = listOf(10,20,30,42)

for (v in numbers) {
  println(v)
}
```

```
val numbers = listOf(10,20,30,42)

val it = numbers.iterator()
while (it.hasNext()) {
  val v = it.next()
  println(v)
}
```

```
// Swift: iterate over range

for v in 1...1000000 {
  print(v)
}
```

```
var range = 1...1000000

var it = range.makeIterator()
while true {
  var v = it.next()
  if (v == nil) { break }
  print(v)
}
```

How Are Iterators Implemented?

There are two primary approaches for creating iterators:

With Traditional Classes

An iterator is an object, implemented using regular OOP classes.

"True Iterators" aka Generators

An iterator is implemented by using a special type of function called a generator.

```
class OurIterator:
  def __init__(self, arr):
    self.arr = arr
    self.pos = 0

  def __next__(self):
    if self.pos < len(self.arr):
      val = self.arr[self.pos]
      self.pos += 1
      return val
    else:
      raise StopIteration

class ListOfStrings:
  def __init__(self):
    self.array = []

  def add(self,val):
    self.array.append(val)

  def __iter__(self):
    it = OurIterator(self.array)
    return it
```

To create the iterator, we need to pass in the private array it's going to

Our iterator then initializes its position to the start of the array.

When the iterator hits the end, __next__() throws an exception.

Ok, first let's define our container class.

Python uses the exception to determine when the loop is complete.

```
nerds = ListOfStrings()
rds.add("Carey")
rds.add("David")
 add("Paul")

for n in nerds:
  print(n)
```

Once we finish, we can use this syntax to iterate over our list!

plement an iterator object
mple List class in Python.

efine an __iter__()
n iterator object.

Our iterator class must have a method called __next__()
e value of the item pointed to by the
ces the iterator, and returns the value.

```
import java.util.Iterator;

class ListOfStrings implements Iterable<String> {
  private String[] array;
  private int numItems = 0, maxSize = 100;

  public ListOfStrings()
    { array = new String[maxSize]; }
  public void add(String val)
    { array[numItems++] =
  @Override public Iterat
    Iterator<String> it =
    return it;
  }

  class OurIterator imple
    private int iteratorI
    @Override public bool
      { return iteratorIn
    @Override public Strin
      { return array[iteratorIndex++]; }
  }
}
```

Let's define a container of Strings class (like a vector)!

To be iterable, it must implement Java's Iterable interface.

Our cla… item…

If we like, we can use the iterator notation explicitly…

Or Java can use its syntactic sugar for us, and let us use built-in iteration syntax!

Since OurIterator is a nested class, it can access the members of the outer class (ListOfStrings).

```
public void testOurList() {
  ListOfStrings nerds =
    new ListOfStrings();

  nerds.add("Carey");
  nerds.add("David");
  nerds.add("Paul");

  Iterator it = nerds.iterator();
  while (it.hasNext()) {
    String value = (String)it.next();
    System.out.println(value);
  }
}
```

Let's Implement a Range Class in Python

```
class OurIter:  # iterator
  def __init__(self, start,
    self.cur = sta
    self.end = end

  def __
    if se
      val
    e
      raise
```

```
class OurRange:  # ite able c
  def __init__(self, end):
    self.end = end

  def __iter__(self):
    iter = OurIter(0, self.end)
    return iter
```

As before, when we reach the end of the range, we raise an exception.

Let's see our iterator class!

Its constructor sets the current position (cur) to the start of the range, and remembers the end.

And then it advances to the next value in the range.

Our constructor simply stores away the upper-end of the range.

Now let's implement a Range class and its iterator!

```
for i in OurRange(5):  # from [0,5)
  print(i)
print("Done!")
```

```
tmp = OurRange(5)
    tmp.__iter__()

True:
iter.__next__()
```

__iter__() method that to get a new iterator.

Finally, here's what the syntactic sugar is hiding!

How Are Iterators Implemented?

With Traditional Classes

An iterator is an object, implemented using regular OOP classes.

"True Iterators" aka Generators

An iterator is implemented by using a special type of function called a generator.

Generator Ch

So what's a generator?
Let's find out!

```
def foo(int n):
  for i in range (1, n):
    print(f'i is {i}')
    pause
    print('woot!')
```

n 4
i 1 2 3

This pauses our function – its instruction pointer is saved along with all variables in the closure.

```
def foo(n):
  for i in range (1, n):
    print(f'i is {i}')
    pause
    print('woot!')

def main():
  p = create_pausable(func=foo, n=4)
  print('CS')
  start(p)
  print('is')
  resume(p)
  print('cool!')
  resume(p)
```

p

This creates a closure for ction, starting used state.

This starts running our paused function.

This resumes our here.
paused function.

Consider the
left, writte

What do you think this will print?

```
CS
i is 1
is
woot!
i is 2
cool!
woot!
i is 3
```

Generators

Our foo() function is called a "generator."

A generator function can be paused and resumed, with its state (variables, instruction pointer) saved between calls!

```
def foo(n):
  for i in range (1, n):
    print(f'i is {i}')
    yield
    print('woot!')

def main():
  p = foo(4)
  print('CS')
  next(p)
  print('is')
  next(p)
  print('cool!')
  next(p)
```

In Python, you create a generator object by simply calling the generator function with its parameters.

We start our generator by calling

Even though you're calling it like a function, this really creates an iterator/closure for the function.

ee the real syntax in Python now.

Generator Challenge, Part 2

```
def bar(a, b):
  while a > b:
    yield a
    a -= 1

f = bar(3, 0)
print('Eat prunes!')
t = next(f)
print(t)
t = next(f)
print(t)
t = next(f)
print(t)
print(f'Explosive diar
```

This not only pauses the generator, but outputs the

The initial call to the generator returns an iterator, implemented using a closure.

So every time we call next(), we're asking the iterator to generate the next value in the sequence.

The statement that called next() on the generator can then ge

Each call to next() resumes the generator until we hit the next yield.

This generator uses yield in a slightly different way.

What do you think this will print?

```
at prunes!
2
1
Explosive diarrhea!
```

This is a more idiomatic use of a generator – they're used to "generate" a sequence of values that can be retrieved one at a time – the sequence might be finite, or infinite!

Looping Over

Academic Robot Says:

"A generator function may be called a true iterator, but it's really an iterable object!"

```
def our_range(a, b):
  while a > b:
    yield a
    a -= 1
```

Iterable object

That's LIT!

Answer: When the generator finally exits without a yield, this signals t

How does Python know when the loop should end?

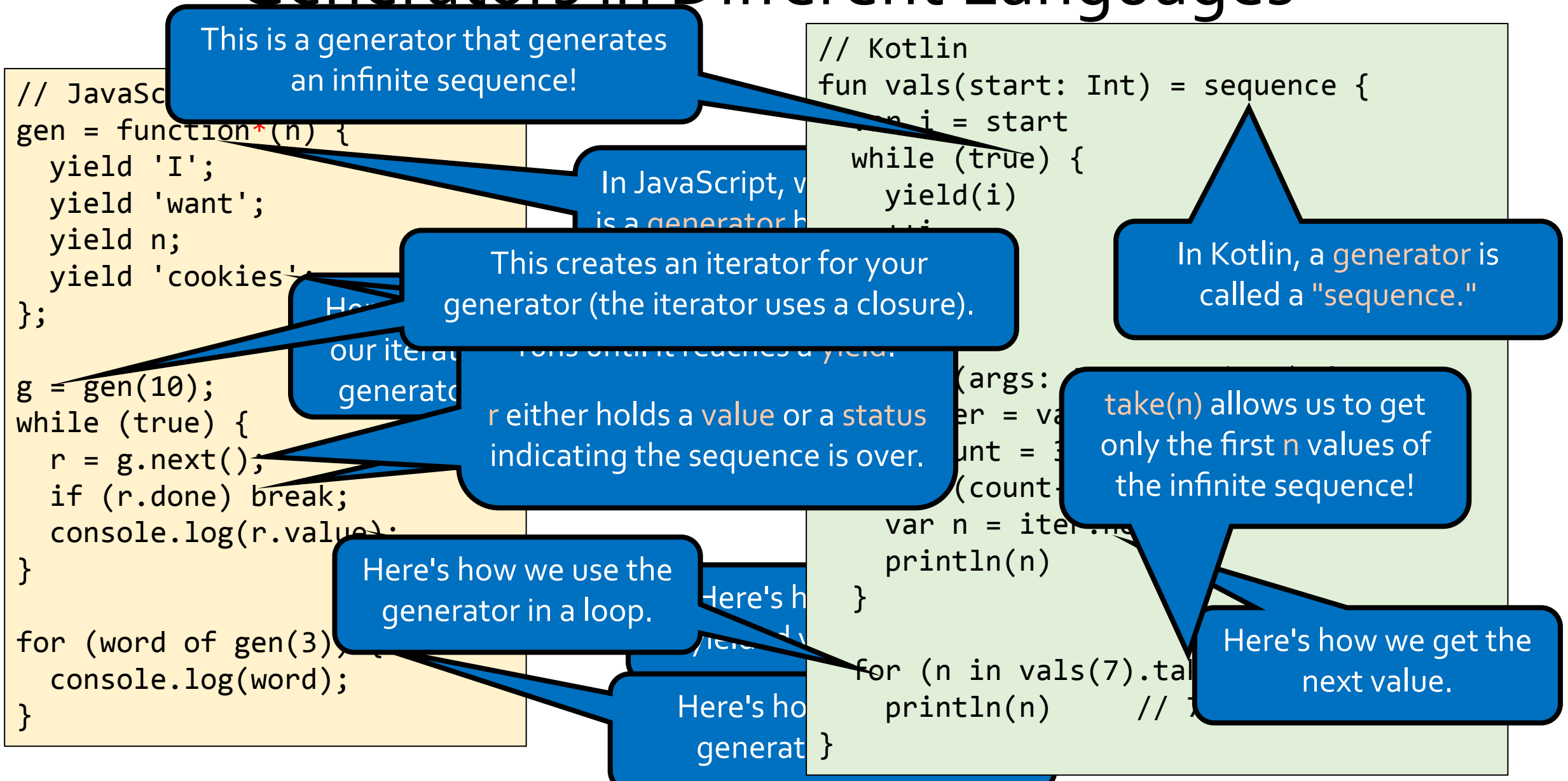
An iterator!

Since a generator is an iterable object, and produces an iterator...

u can use it like any other iterable object in loops!

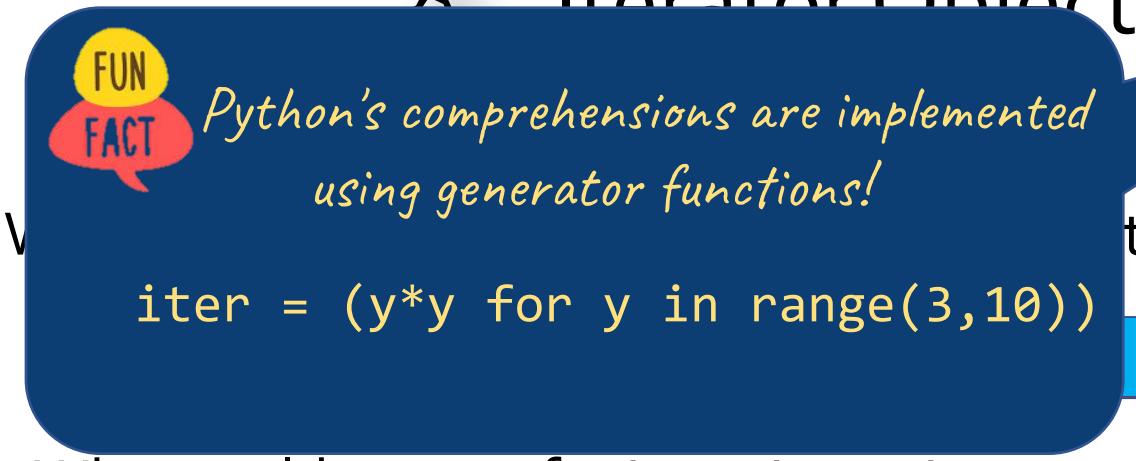
Another name for a generator function is a "true iterator."

Generators in Different Languages



Let's see some generators in different languages.

Iterator Objects and Generators



W t you can't do with a generator, and visa-versa?

Why would you prefer iterating using a generator vs. iterating over a list of the same size?

```
for i in range(1,10000000):
  print(i)
  if some_condition(i) == True:
    break
```

```
lst = [1, 2, 3,..., 10000000]
for i in lst:
  print(i)
  if some_condition(i) == True:
    break
```

Which is easier: Creating the code for an iterator object or a generator so they can do an in-order traversal of a BST? Bonus: Try to write either one in one minute!

How Does Iteration Actually Work?

Alright, enough of iterators!

Let's see how to iterate with first-class functions.

These kinds of loops are implemented using "iterators"

```
for n in student_names:
 print(n)
```

```
for i in range(0,10):
  print(i)
```

Loops like these are built using first-class functions

```
fruits.forEach
 { f -> println("$f") }
```

Iteration with First Class Functi

Earlier we saw the following type of looping s

```rust
// Rust
(0..10).for_each(|elem| println!("{}", elem));
```

```rust
let items = vec!["fee","fi","fo","fum"];

items.iter().for_each(|elem| println!("{}", elem));
```

```kotlin
// Kotli
(0..9).f
```

```kotlin
var items =

items.forEach({ elem -> println("$elem") })
```

```ruby
# Ruby
(0..9).each do |elem|
  print elem, " "
```

```ruby
ee","fi","fo","fum"]

do |elem|
    print(elem,"\n")
end
```

Ruby's syntax is a bit weird... But while this looks like a block, it's the same as a lambda!

In many languages, an iterable object will have a forEach() or each() method.

The method will apply the passed-in function to each of the iterable's values.

With this syntax, we're NOT using an iterator!

Instead, we're passing a function as an argument to a forEach()/each() method that loops over the iterable's items!

```swift
// Swift
(0...9).forEach
  { elem in print(elem) }
```

```swift
let items = ["fee","fi","fo","fum"]

items.forEach { elem in print(elem) }
```

Iteration with First Class Functions

Let's build a forEach() method for our earlier Python container:

```
class ListOfStrings:
  def __init__(self):
    self.array = []

  def add(self,val):
    self.array.append(val)

  def forEach(self, f):
    for x in self.array:
      f(x)

yummy = ListOfStrings()
... # add items to yummy

def like(x):
 print(f'I like {x}')

yummy.forEach(like)
```

rEach() method
the items itself...

Of course, we don't need to pass a lambda – we can also pass a regular function.

As you can see, an iterator is not used for this syntax!

Instead, the iterable object iterates over its own items, and calls the provided f().

This is essentially a map() operation like we saw in functional programming!

Iterating With First-class Functions

What are the pros/cons of using forEach() + a first-class function vs. an iterator?

What happens if you delete an item from a container while you're iterating over its items using any of the approaches?

Classify That Language: Iterators

```
class Kontainer implements Iterator {
  private $array = array();

  public function add($val)
    { array_push($this->array,$val); }

  private $pos = 0;

  public function rewind() { $this->pos = 0; }

  public function current()
    { return $this->array[$this->pos]; }

  public function key()   { return $this->pos; }

  public function next()  { ++$this->pos; }

   public function valid()
    { return $this->pos < count($this->array); }
}
```

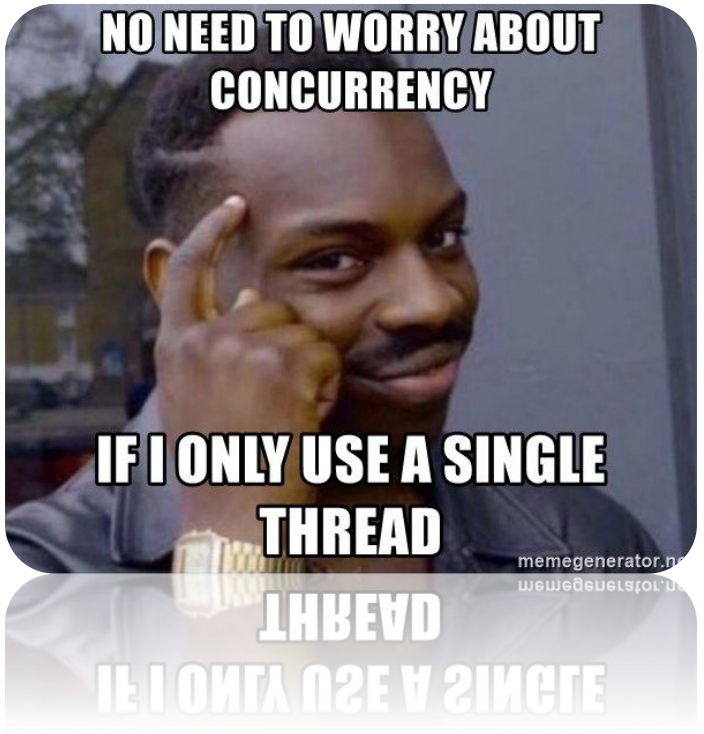
In the class to the left, we have created a container class that supports iteration.

Will this class support nested loops?
If so, why? If not, why not?

```
$letters = new Kontainer;
$letters->add("a");
$letters->add("b");
$letters->add("c");

foreach($items as $i) {
  foreach($items as $j) {
    print( $i . " " . $j . ", ");
  }
}
```

This is PHP!

Concurrency



Many modern programming languages have explicit support for concurrent execution (running multiple things at once).

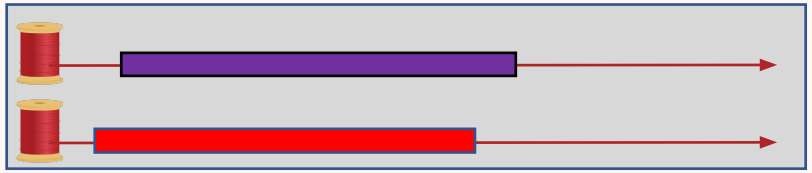
This is a huge domain, so we'll just do a quick survey of a few relevant areas.

Concurrency

Concurrency is a paradigm in which a program is decomposed into simultaneously executing tasks.

Those tasks might…

Run in parallel on multiple cores



Run multiplexed on a single core

Operate on totally independent data
e.g., each sorting a different array

3	5	2	9	4

1	0	7	19	-5

Operate on shared *mutable* data or systems
e.g., two tasks modifying a shared queue

Be launched deterministically as part of the regular flow of a program

```
def solve_a_problem():
 a = run_concurrently task1()
 b = run_concurrently task2()
 use_results_when_done(a,b)
```

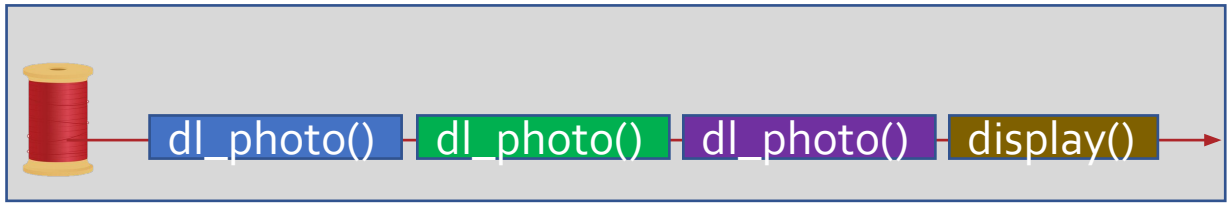
Be launched due to an external event occurring, like a click of a button in a UI

Concurrency

Concurrency can make our programs more efficient!

The old way: Serial Execution

```
def slide_show(url1, url2, url3):
 photos = []
 dl_photo(url1, photos)
 dl_photo(url2, photos)
 dl_photo(url3, photos)

 display(photos)
```



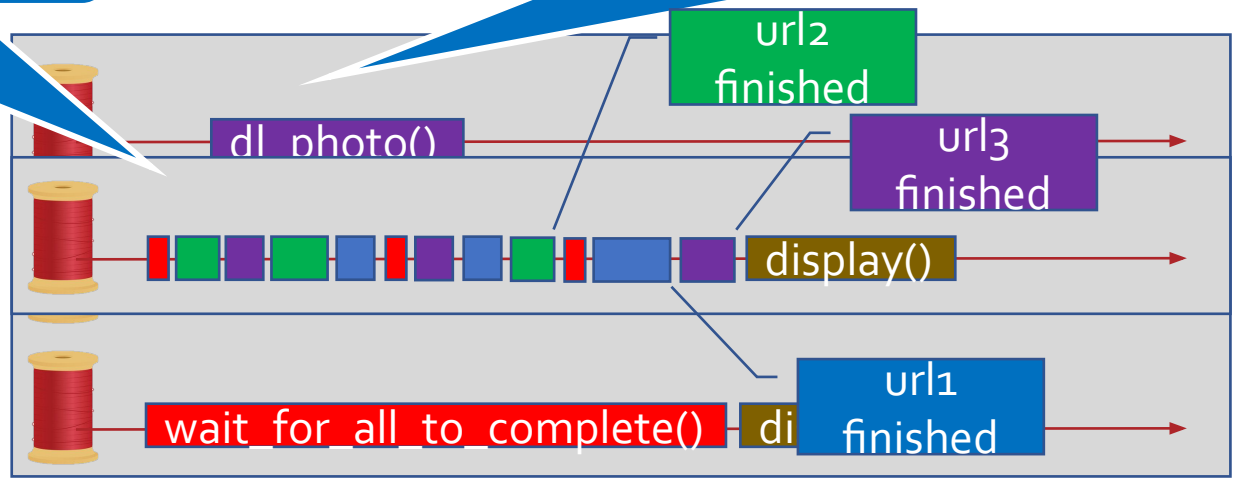
By the way, this is pseudocode – not correct syntax!

Or in a single thread, like this.

nt Execution

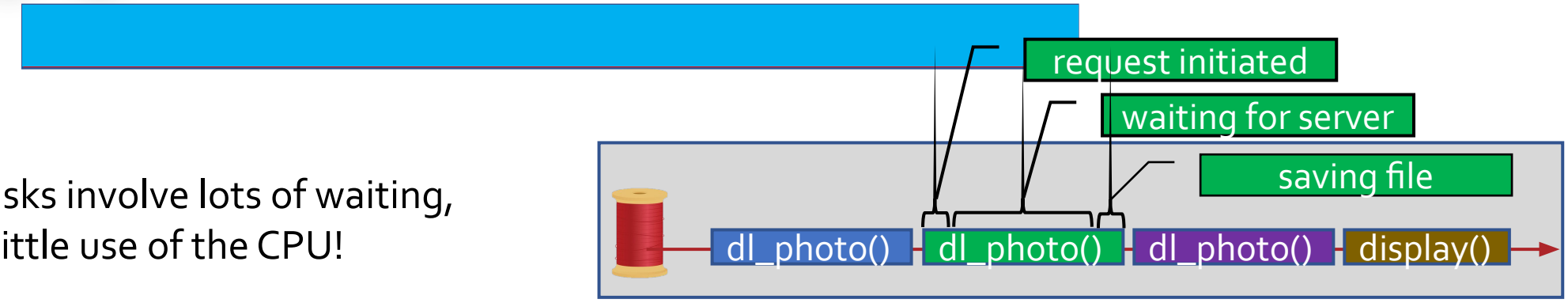
The concurrently-running tasks could run in parallel like this…

```
def slide_show(url1, url2, url3):
 photos = []
 a = run_concurrently dload_photo(url1, photos)
 b = run_concurrently dload_photo(url2, photos)
 c = run_concurrently dload_photo(url3, photos)
 wait_for_all_to_complete(a,b,c)

 display(photos)
```



Concurrency

If a concurrent program only runs on a single core, why would it be any faster than a program that runs serially?

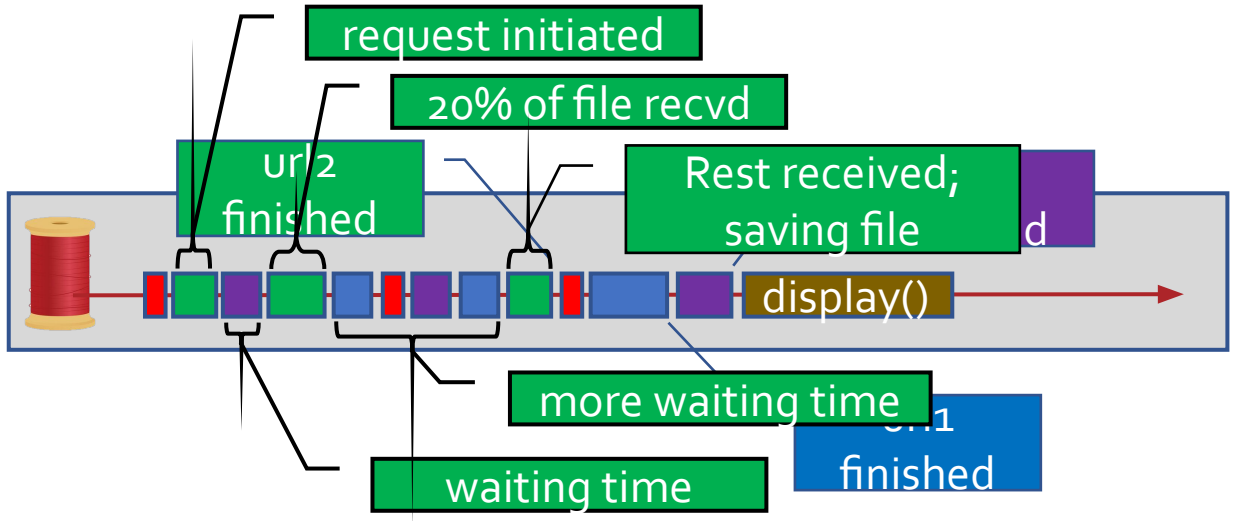


Answer: Some tasks involve lots of waiting, but make little use of the CPU!

Consider downloading a photo!

90% or more of the time might be waiting for the server to send the file!

During that waiting time, a concurrent program can use the CPU for other useful tasks!



karma would be an example of a mutable variable that's shared between multiple threads.

Academic Robot Says:

"A key goal of concurrency is *determinism*: ensuring that code produces the same result every time it runs (regardless of how its tasks are scheduled)!"

register1

```
// karma++ i...ssemb...
mov register1, [ka...ma]
inc register1
mov [karma], register1

void do_good_deeds() {
  for (int i=0;i<100000;++i
    karma++;
}

void be_naughty() {
  for (int i=0;i<100000;++i)
    karma--;
}

...
```

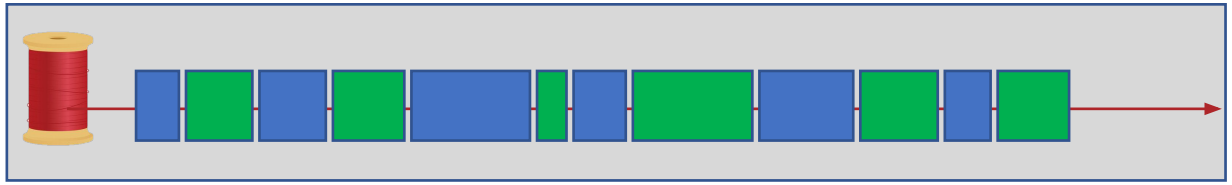
karma 1

Our good karma thread completely overwrites the change made by the bad karma thread!

We might then pause again here, to run our other thread.

What will the value of karma be if I run both functions concurrently until completion? Why?

'thread" of execution can be interrupted
:he other, it's impossible to know!



Any time two threads of execution use "shared mutable state", this kind of thing can happen.

This is called a "race condition."

Concurrency With Shared Mutable State

```
class Karma extends Thread {
  private static int karma = 0

  static void do_good_deeds() {
    for (int i=0;i<100000;++i)
      synchronized(Karma.class) { karma++; }
  }

  static void be_naughty() {
    for (int i=0;i<100000;++i)
      synchronized(Karma.class) { karma--; }
  }

  public static void main(String args[])
    ... // Code to start two threads and
    ... // do_good_deeds() and be_naughty()

    System.out.println(karma);
  }
}
```

The synchronized keyword ensures that only one of these two blocks can run at a time.

Most modern languages now have built-in language features to make it safer to use shared mutable state.

For example, Java's synchronized keyword can be used to limit access to a mutable variable to a single thread at a time.

ese features in any know they exist.

But be careful – it's still difficult to get shared mutable state right (and efficient)!

Models for Co

There are two primary approaches

Academic Robot Says:

"Don't forget to mention there are hybrids, Carey!"

Multi-threading Model

A program creates multiple "threads" of execution that run concurrently (potentially in parallel).

The programmer explicitly "launches" one or more functions in their own threads, and the OS schedules their execution across available CPU cores.

```
// Multi-threaded program (pseudocode)
void handle_user_purchase(User u, Item i) {
  if (bill_credit_card(u) == SUCCESS) {
    create_thread(schedule_shipping(u, i));
    create_thread(send_confirm_email(u, i));
  }
}
```

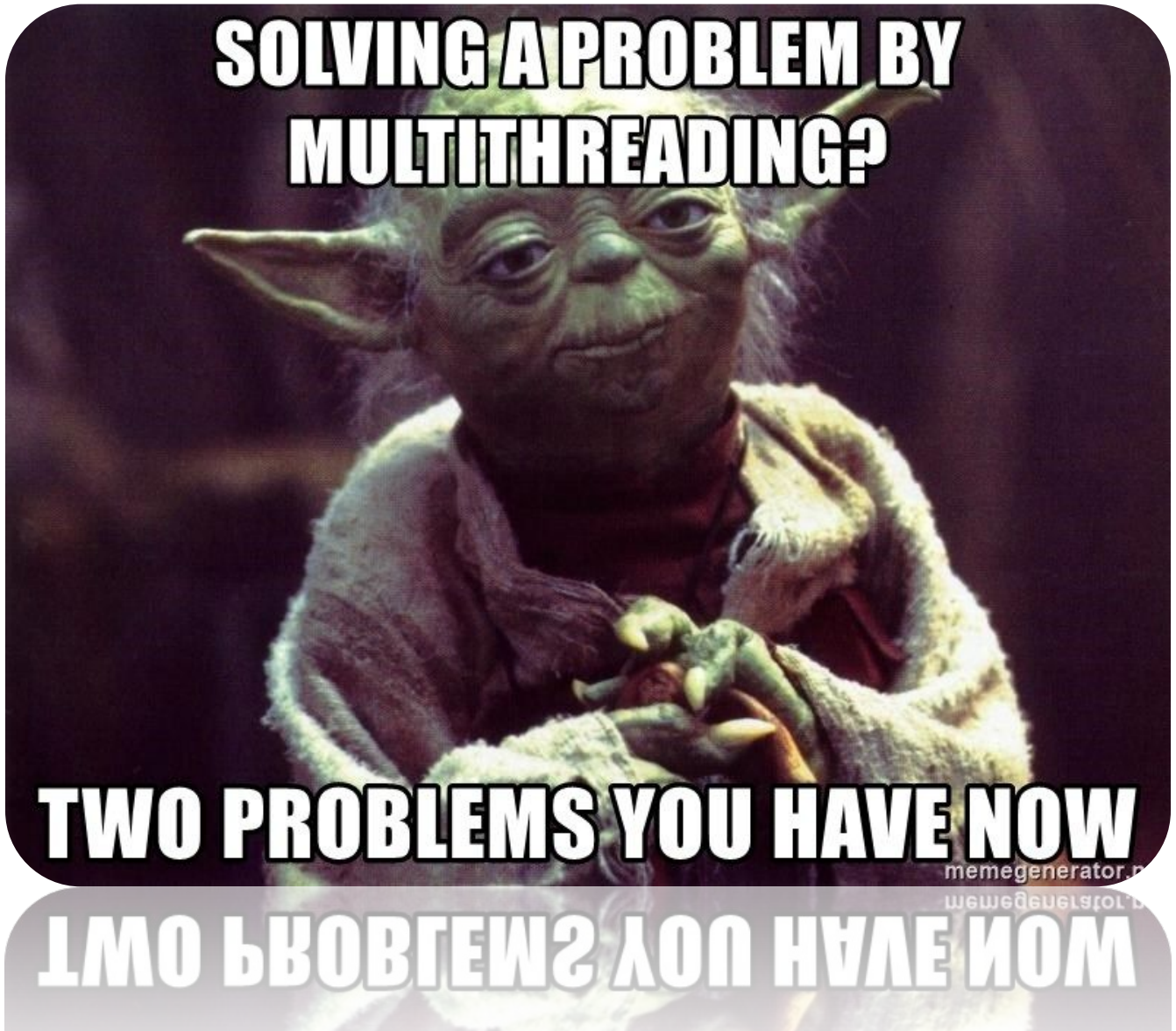
Event Driven Model

A program consists of a queue of functions to run, and an infinite loop that dequeues and runs each function from the queue, one after the other.

When an event occurs (e.g., user clicks a button) the event results in a new function f() being added to the queue which eventually runs and handles the event.

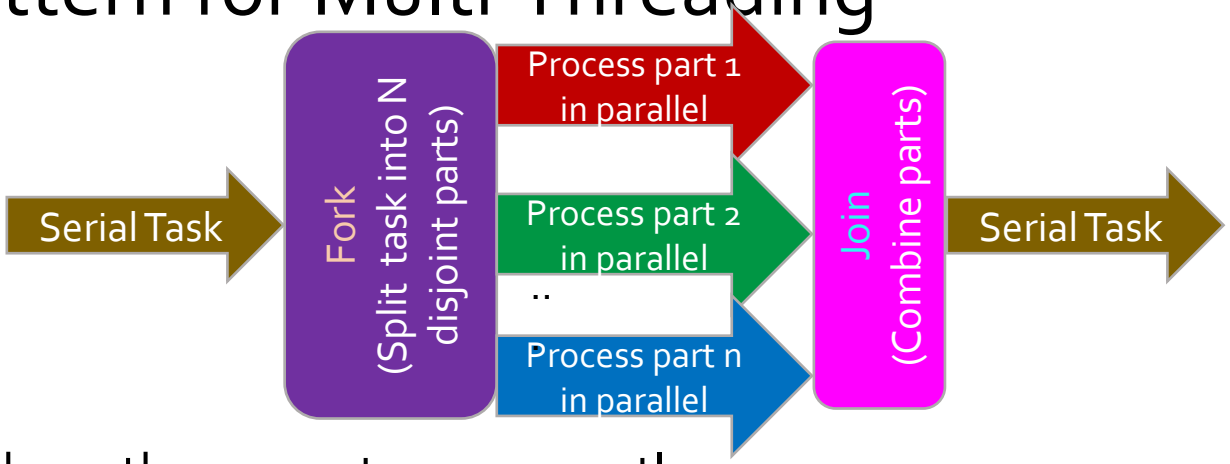
```
// Event-driven program (pseudocode)
function process_payment() { ... }

void setup_event_associations() {
 button = create_new_button("Pay Now!");
 button.set_func(ON_CLICK, process_payment);
}
```

Multi-Threading



Fork-Join: A Common Pattern for Multi-Threading

The "fork-join" pattern is basically a ... and conquer.



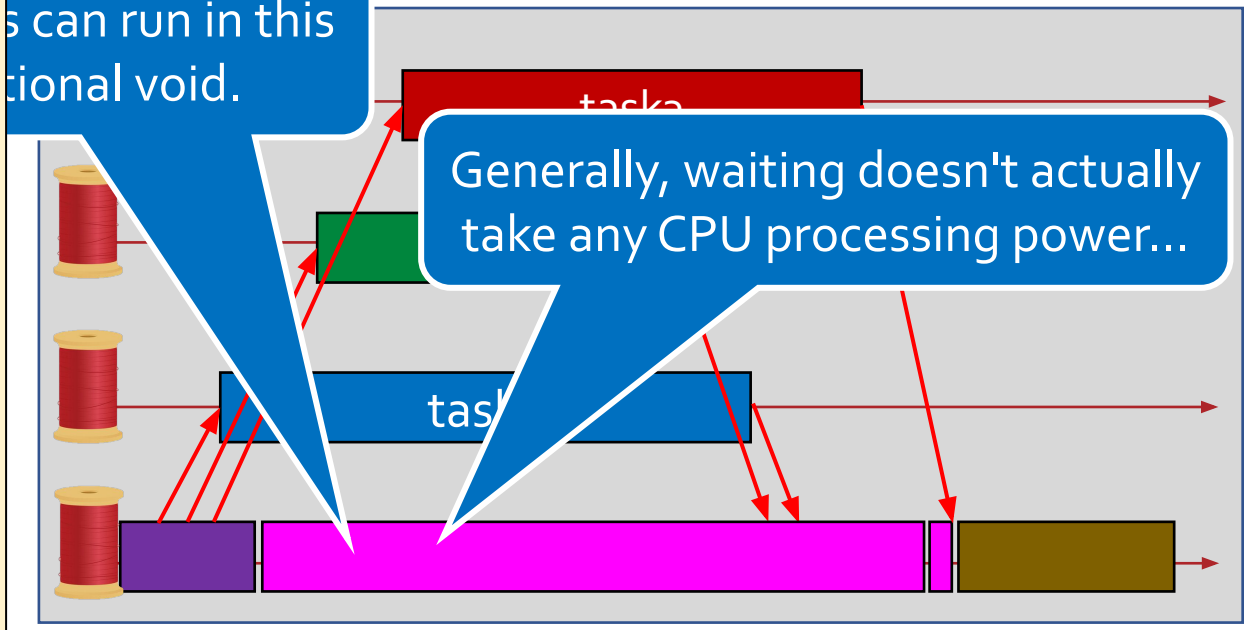
A parallel sort would be a good example of a problem suitable for fork-join.

First, we "fork" one or more tasks so they execute concurrently...

Second, we wait for all those tasks to complete (aka "join") and then proceed.

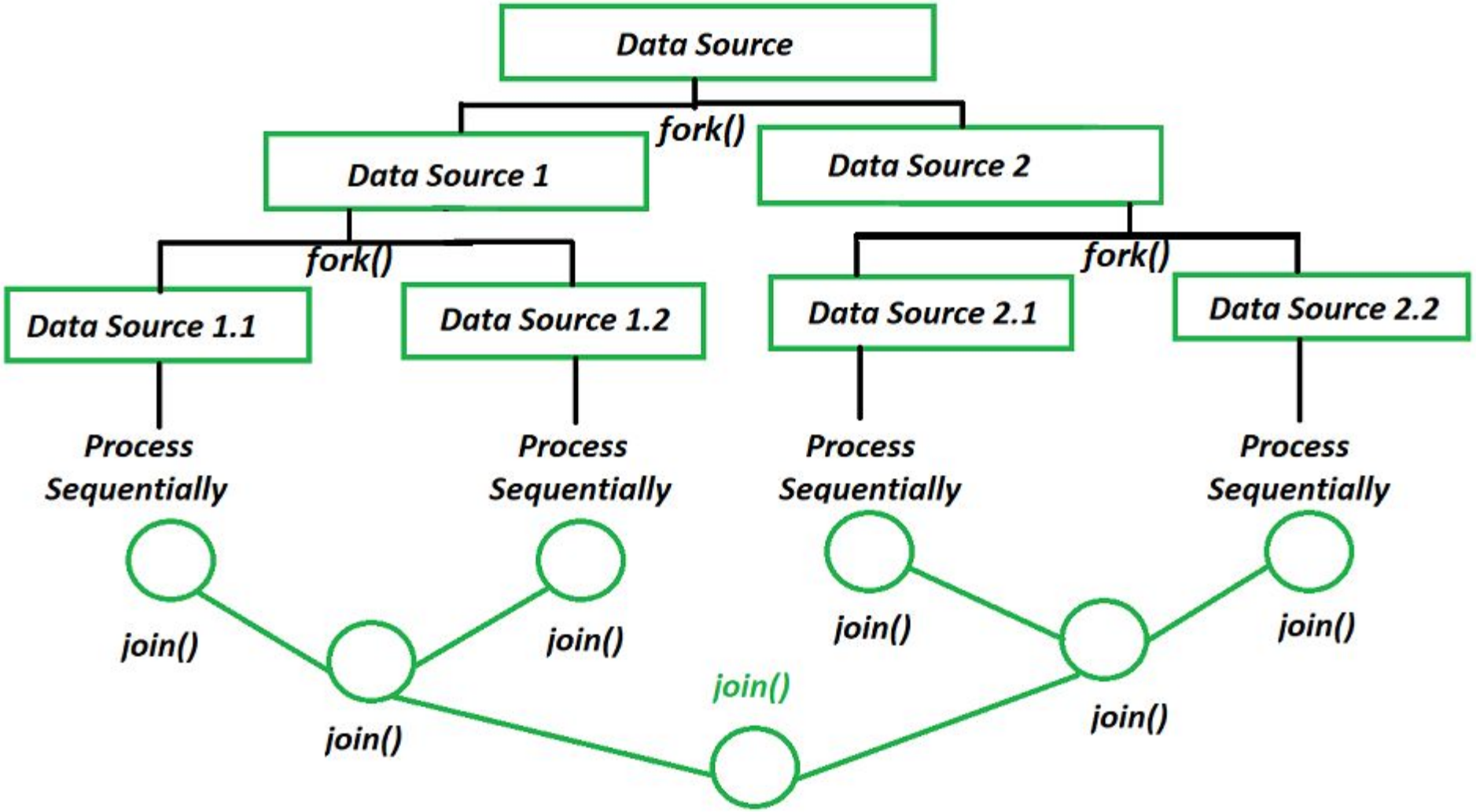
```
function sort_in_parallel(array, n) {
  t1 = run_background(sort(array[0:n/3]));
  t2 = run_background(sort(array[n/3:n*2/3));
  t3 = run_background(sort(array[n*2/3:n]));

  wait_for_all_tasks_to_finish(t1, t2, t3);
  merge_sorted_subarrays(array);
}
```

...s can run in this ...tional void.



Generally, waiting doesn't actually take any CPU processing power...

Fork-Join is Often Used Recursively



Source: GeeksForGeeks

Fork-Join In Different Lang

```cpp
// C++
#include <thread>

void task1(int n)
void task2(int n)
void task3(int n)  {

int main() {
  std::cout << "F rking threads!\n";
  std::thread t1(task1, 10);
  std::threa
  std::threa

  // do othe

  t1.join();
  t2.join();
  t3.join();
  std::cout << "All
}
```

```
// C#
System;
System
System

class Progra

m) { ...
m) { ...
m) { ... }

public static in(string[] ar ) {
  int r1 = 0,  =  3 = 0;
  Console.Wri eLine("Fo ing threads ");
  var t1 = Task.Run(() => r1 = task1(10));
  = Task.Run(() => r2 = task2(20));
  = Task.Run(() => r3 = task3(30));
  aitAll(t1, t2, t3);
  e.WriteLine("Joined: {0}, {1}, {2}",
                r1, r2, r3);
}
```

To la
bac
creat

We could have done a similar thing here to get the results if we liked.

The lambda function then calls our desired task function with the proper arguments, and then gets the result.

To fork, we use task.Run(the passed-in lambda fun

Finally, since we stored the result of our tasks in local variables…

We use Task.WaitAll() to join multiple tasks at once.

To join, we simple call the thread.join() method. This blocks the caller until the thread finishes.

after we join the results.

Fork-Join In Different Languages: Python

```
# Python
import threading

def task(n):
  while n > 0:    # do some compu
    n = n - 1

print("Forking threads!")
t1 = threading.Thread(target=task, args=(100000000,))
t2 = threading.Thread(target=task, args=(100000000,))
t3 = threading.Thread(target=task, args=(100000000,))
t1.start()
t2.start()
t3.start()

# do other processing here...

t1.join()
t2.join()
t3.join()
print("All threads joined!")
```

Arguments to a task are passed in via a tuple.

In Python, we must create a thread object, and then do thread.start() before it runs.

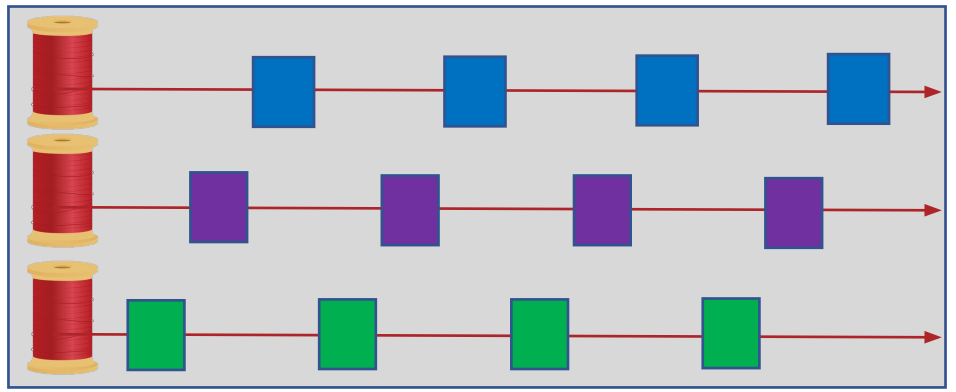
And here's how we join.

Assuming looping 100 million times takes 5s, how long will this program take to run on a multicore PC?

We'd expect all three tasks to run in parallel… taking 5s total. But it takes 15s!

Why? Because when each Python thread runs, it claims exclusive access to Python's memory/objects.

So only one thread generally does computation at a time!



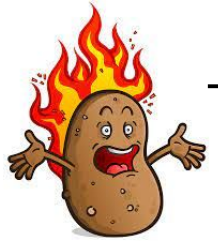
Fork-Join In Different Languages: Python

```
# Python
import threading

def task(n):
  while n > 0:
    n = n - 1

print("Forking threads!")
t1 = threading.Thread(target=task, args=(100000000,))
t2 = threading.Thread(target=task, args=(100000000,))
t3 = threading.Thread(target=task, args=(100000000,))
t1.start()
t2.start()
t3.start()

# do other processing here...

t1.join()
t2.join()
t3.join()
print("All threads joined!")
```

Why? Python's garbage collection system was never designed to be thread-safe!

So, what's going on?

Python has something called a "Global Interpreter Lock" or GIL.



The GIL is like a hot potato – it can only have one owner at a time.

Once a thread takes ownership of the GIL it's allowed to read/write to Python objects.

After a thread runs for a while, it releases the GIL (tosses the potato) to another thread and gives it ownership.

If a thread is waiting for the GIL, it simply falls asleep until it gets its turn.

Fork-Join In Differ

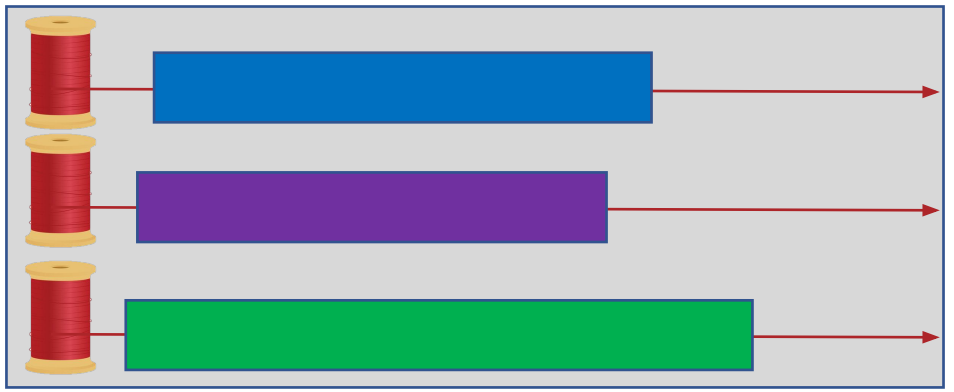
Academic Robot Says:

"In addition to I/O ops, many external C++ libraries (e.g., pytorch) have operations that can also run in parallel!"

```
# Python
import threading

def task(symbol):
  response = urllib2.urlopen('www.stocks.com/'+symbol)
  html = response.read()

print("Forking threads!")
t1 = threading.Thread(target=task, args=("goog",))
t2 = threading.Thread(target=task, args=("amzn",))
t3 = threading.Thread(target=task, args=("bby",))
t1.start()
t2.start()
t3.start()

# do other processing here...

t1.join()
t2.join()
t3.join()
print("All threads joined!")
```

Answer: I/O operations like downloading data from the web or saving a file to disk DON'T need the GIL to run!

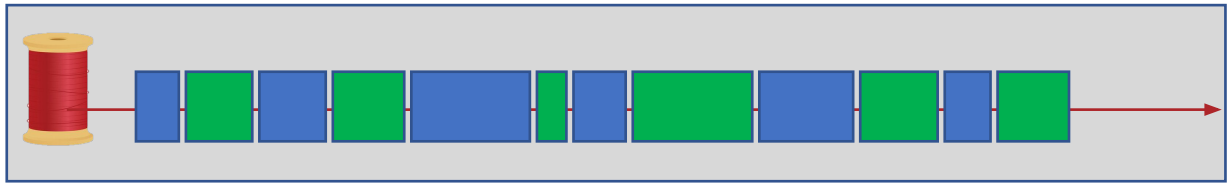
So these kinds of operations can still make progress if launched in the background!



UI Programming and Concurrency

When we're building user interfaces, concurrency works a bit differently.

It might be surprising, but most UIs (Windows, MAC, iPhone, Android, browsers) are single-threaded and "event-based"!



Let's see how event-based concurrency works next!

But First… Callback Functions

A callback function is a function that is called when an "event" occurs.

The event could be a click of a UI button, a timer expiring, or a completed payment transaction.

The callback performs a follow-up action that hand

```
function set_up_ui() {
  payment_button = new Button("Pay now!");
  payment_button.set_callback(ON_CLICK, call_upon_click)
  add_button_to_ui(payment_button);
}

function call_upon_click() {
  pay_obj = new PaymentProcessor();
  pay_obj.set_callback(call_upon_payment_done);
  pay_obj.initiate_payment_in_background();
}

function call_upon_payment_done(result) {
  if (result == SUCCESS) update_ui("Payment acc
                    else update_ui("Card denied
}
```

This says: "Call the call_upon_click() function if/when the user clicks the payment button.

And instructs it to call this callback when it eventually completes.

This initiates the payment in the background… It might take a while…

For example, a callback handling a button click could initiate a credit-card purchase…

and the callback that's triggered when the purchase completes would update the webpage with esult.

A S

```
<html>
<body>
<h1>Fortune Teller</h1>
<p id="fort">What does the future hold?</p>
<button onclick="callbk();">Get Fortune</button>

<script>
  set_background_color(BLACK);
  play_mp3("spooky_music.mp3");

  function callbk() {
    set_element("fort", "You'll ace CS131!");
  }
</script>

</body>
</html>
```

der this webpage written in pseudo HTML and JavaScript.

It defines a text field where we'll show our fortune.

This specifies a "handler" or "callback" function for our button.

This page also has top-level statements that run when the page initially loads.

s executed when licked.

In this example, our handler updates the "fort" field with the user's fortune.

Web Browser in a Slide

```
class Browser:
```

While traditional programs run from top-to-bottom, programs with UIs use a different model.

Why? UIs contain elements (e.g., buttons, sliders) that the user can interact with in any order!

So UIs use something called an Event Loop to control their execution.

Let's look at a high-level model for a web browser to see how the Event Loop works.

Web Bro

```
class Browser:
  def __init__(self, page):
    self.objs_on_page = self.get_all_objs(page)
    self.run_queue = self.extract_stmts(page)
```

The browser keeps track of anything in the UI that can be interacted with by the user.

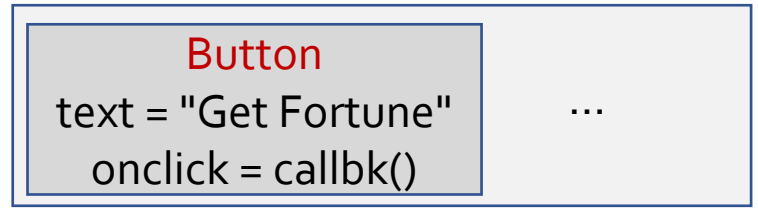
The browser starts by queuing up all of the top-level statements on the webpage for execution.

```
<html>
<body>
<h1>Fortune Teller</h1>
<p id="fort">What does the future hold?</p>
<button onclick="callbk();">Get Fortune</button>

<script>
  set_background_color(BLACK);
  play_mp3("spooky_music.mp3");

  function callbk() {
    set_element("fort", "You'll ace CS131!");
  }
</script>

</body>
</html>
```

The browser manages two sets of items:

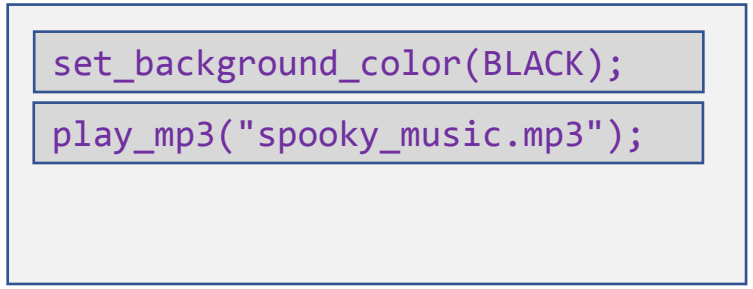
bjects on a page
liders, etc.).

objs_on_page



It maintains a queue of statements to execute from the page.

run_queue



Web Browser in a Slide

```
class Browser:
  def __init__(self, page):
    self.objs_on_page = self.get_all_objs(page)
    self.run_queue = self.extract_stmts(page)

  def event_loop(self):
    while True:
      self.handle_events()
      self.run_next_statement()
```

Now let's see our event loop – it runs an infinite loop that does two things.

First, it checks to see if the user interacted with the page's elements, and if so, deals with this.

Second, it gets the next statement from the queue (if there are any) and runs it.

The loop runs over and over, handling events, and running the next queued statement…

r in a Slide

```
...
<button onclick="callbk();">Get Fortune</button>

<script>
  set_background_color(BLACK);
  play_mp3("spooky_music.mp3");

  function callbk() {
    set_element("fort", "You'll ace CS131!");
  }
</script>
...
```

```
def handle_events(self):
  for obj in self.objs_on_page:
    if obj.event_occurred():
      func = obj.get_handler()
      self.run_queue.append(func)
```

How do we handle user-interaction events?

We cycle through each of the UI elements on the screen to see if any have been interacted with…

If an element has experienced an event (e.g., a click)…

we ask it for its handler (aka callback) function…

But we DON'T call the handler immediately!!!

Instead, we queue the function up for later execution!

run_queue

```
set_background_color(BLACK);
play_mp3("spooky_music.mp3");
callbk();
```

Web Browser in a Slide

```
class Browse
  def __init
    self.obj
    self.run

  def event_
    while Tr
      self.h
      self.r

  def handle_events(self):
    for obj in self.objs_on_page:
      if obj.event_occurred():
        func = obj.get_handler()
        self.run_queue.append(func)

  def run_next_statement(self):
    if len(self.run_queue) > 0:
      func = self.dequeue()
      func()
```

Finally, let's see how we run statements.

We just check to see if the queue has pending statements to run...

And if so, we dequeue the next statement and run it to completion!

run_queue

```
set_background_color(BLACK);
play_mp3("spooky_music.mp3");
callbk();
```

Web Browser in a Slide

```
class Browser:
  def __init__(self, page):
    self.objs_on_page = self.get_all_objs(page)
    self.run_queue = self.extract_stmts(page)

  def event_loop(self):
    while True:
      self.handle_events()
      self.run_next_statement()

  def handle_events(self):
    for obj in self.objs_on_page:
      if obj.event_occurred():
        func = obj.get_handler()
        self.run_queue.append(func)

  def run_next_statement(self):
    if len(self.run_queue) > 0:
      func = self.dequeue()
      func()
```

The Event Loop runs over and over…

Checking for new events…

Potentially adding items to the run queue…

Then running the next item in the run queue until completion…

Web Browser in a Slide

```
class Browser:
  def __init__(self, page):
    self.objs_on_page = self.get_all_objs(page)
    self.run_queue = self.extract_stmts(page)

  def event_loop(self):
    while True:
      self.handle_events()
      self.run_next_statement()

  def handle_events(self):
    for obj in self.objs_on_page:
      if obj.event_occurred():
        func = obj.get_handler()
        self.run_queue.append(func)

  def run_next_statement(self):
    if len(self.run_queue) > 0:
      func = self.dequeue()
      func()
```

I/O is also tracked by the event handler just like UI events!

When they finish, the ev handler queues up a callback to run on the main thread.

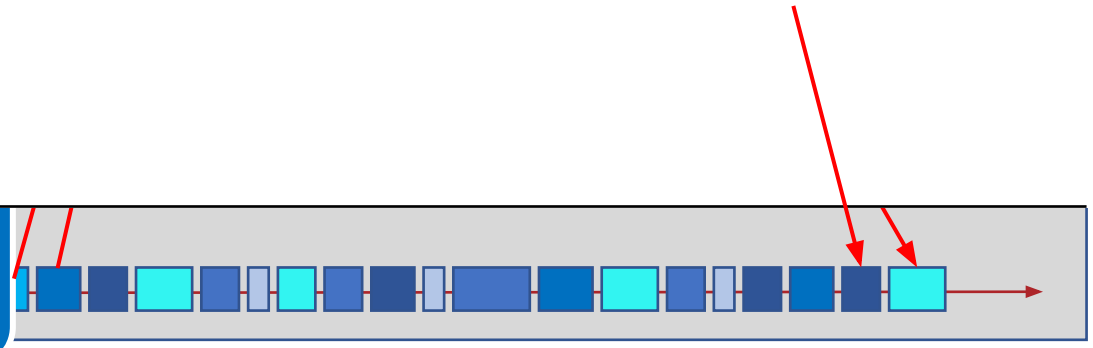
But wait, does ALL browser activity run in the event loop?

OK, maybe I oversimplified.

By default, all user-supplied UI code (e.g., Javascript) DOES run on the event loop…

But things like I/O (e.g., downloading data, querying a DB) DO run on background threads!

In addition, you can also launch your own background threads, but there are caveats.



Event Loops, Callbacks and I/O: Example

```
<html><body>
<div>The current weather is:</div>
<div id="weather">No weather data</div>

<script>

function update_weather_in_ui
  /* code to update the weat

function download_weather() {
  var req = new WebR
  req.callback_on_co
  req.set_url("https
  req.launch_in_back
}

download_wea
set_backgrour

</script>
</body></html>
```

A few seconds later, our web request finishes and the event triggers.

We then dequeue the next function call and run it to completion.

Our function continues running immediately…

web request in
nd registers it
t for the page.

aded, all
ents are
ueue.

www.weather.com

The current weather is:
72 and cloudy

function to the run queue.

XTMLH equest
on_co pletion =
update_weather_in_ui()

run_queue

```
update_weather_in_ui();
set_background_color(BLUE);
```

Event Loop Challenge

```
class Browser:
  def __init__(self, page):
    self.objs_on_page = self.get_al
    self.run_queue = self.extract_s

  def event_loop(self):
    while True:
      self.handle_events();
      self.run_next_statement();

  def handle_events(self):
    for obj in self.objs_on_page:
      if obj.event_occurred():
        func = obj.get_handler()
        self.run_queue.append(func)

  def run_next_statement(s
    if len(self.run_queue)
      func = self.dequeue(
      func()
```

The event handler can't run and process UI events like clicks, etc.

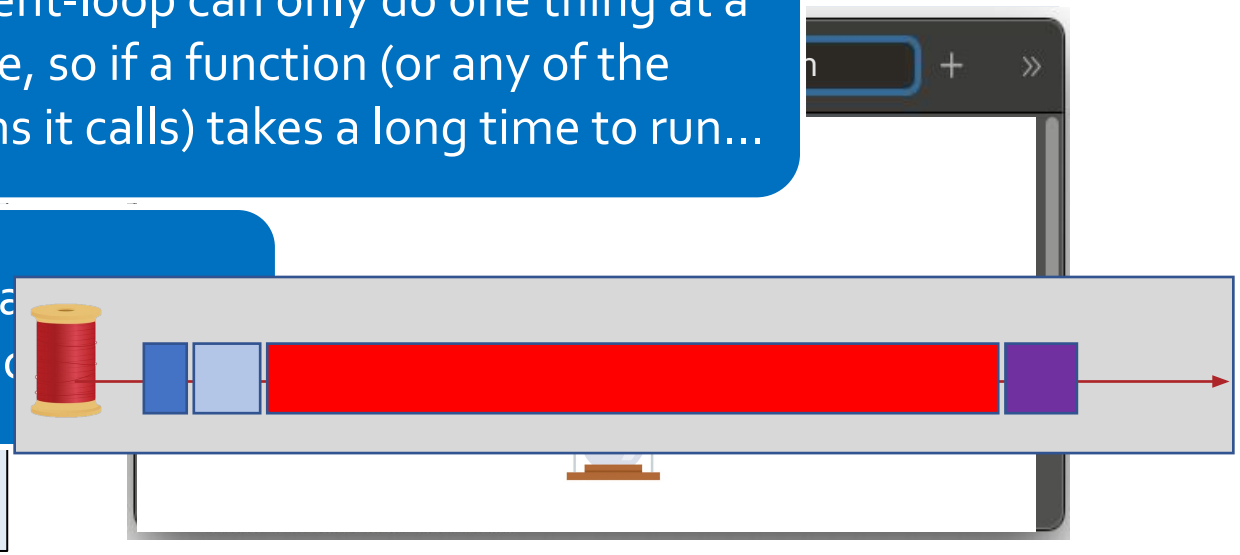
e of the statements in the run " and takes a long time to run?

run_queue

```
run_billion_iteration_loop();
handle_button_click();
```

The event-loop can only do one thing at a time, so if a function (or any of the functions it calls) takes a long time to run...

This causes the UI to ha become unresponsive to



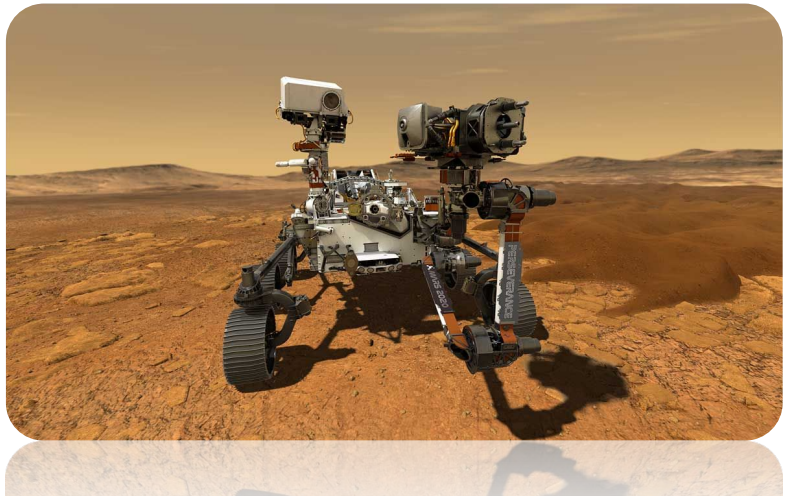
Event Loops are Everywhere

Mobile App Frameworks

Backend Frameworks

Industrial Robotics

Even the Mars Rover!



Event Loops and Sequences of Callbacks

```
<html><body>
<div>The current weather is:</div>
<div id="weather">No weather data</div>

<script>

function update_weather_in_ui()
 { /* code to update the weather field */ }

function download_weather() {
  var req = new WebRequest();
  req.callback_on_completion = update_weather_in_ui;
  req.set_url("https://www.weather.com/...");
  req.launch_in_background();
}

download_weather();
set_background_color(BLUE);

</script>
</body></html>
```

In this example, we saw how to run a long-running op in the background…

And use a callback upon completion to perform the next processing step.

While sometimes we'll just need to run a single background op…

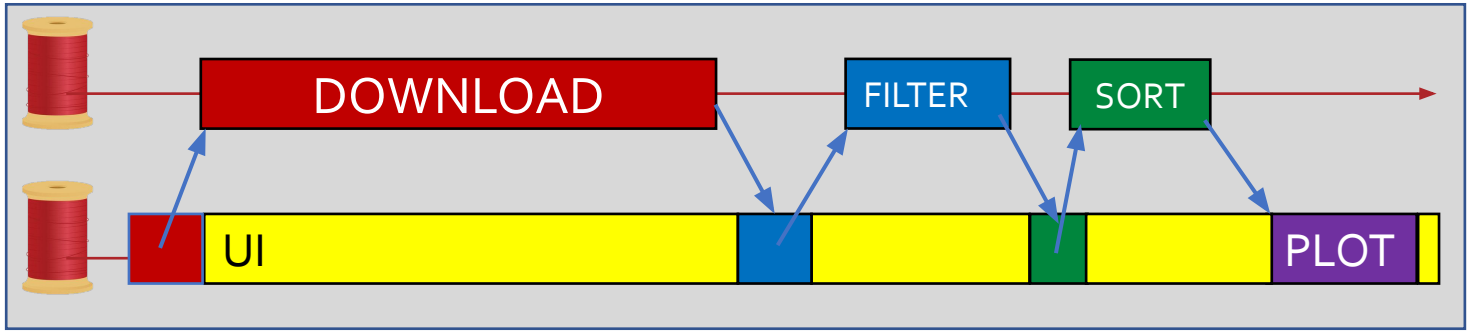
In many cases, we want to chain multiple long-running background ops together into a pipeline.

This lets us accomplish more complex tasks in the background.

Chaining Background Ops Together

Let's say we want to download some data, then filter the data, then sort the data, then update the UI.

We'd like to perform these tasks in the background, while the event loop lets the user interact with the UI...



To make this happen, we provide each phase of the pipeline with a callback function that indicates the next function to run.

The second-to-last stage runs its task in the background...

```
function launch_sort(data, callback) {
  run_in_background {
    sort_data(data);
    add_to_run_queue(callback);
  }
}
```

And once it finishes, it adds its callback function to the run queue.

haining (With Callbacks) Can Get Ugly

going into detail, the syntax to chain multiple background functions
ner with callbacks is pretty ugly and requires the use of lambda functions...

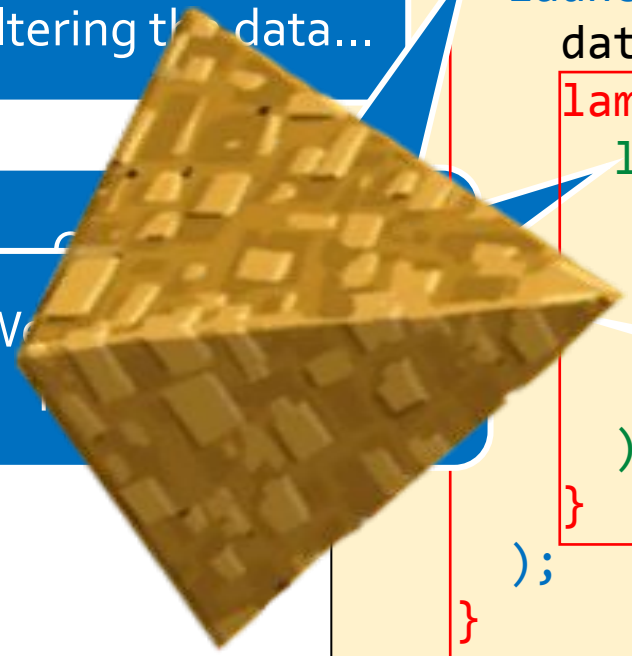
```
function display_sorted_filtered_data(url) {
  launch_download(
    url,
    lambda (data) {
      launch_filter(
        data,
        lambda (fdata) {
          launch_sort(
            fdata,
            lambda (sdata) {
              display_in_ui(sdata);
            }
          );
        }
      );
    }
  );
}
```

Once we finish downloading the data...

Once we finish filtering the data...

We run a callback that sorts the data...

We run a callback that displays the data...



This nesting syntax gets so ugly that it's been given a few names including...

Callback Hell and The Pyramid of DOOM!

Promises: A Cleaner Syntax for Chaining Operations

, and then filter the data, and then sort
lot the data in the UI

ce if ... uld use a simpler syntax than callbacks?

For this to work, each of our functions must be rewritten to use Promise objects.

Once the data has finished downloading...

Once the data has finished filtering...

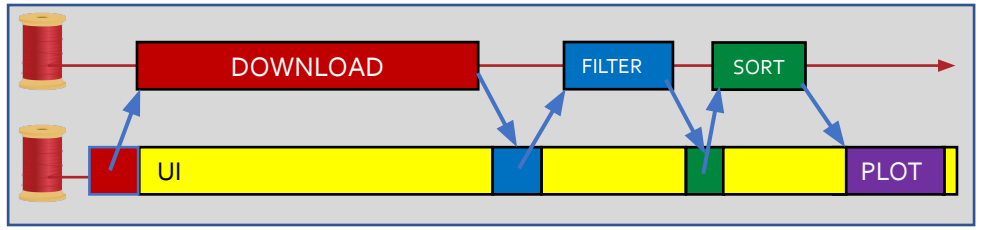
```
function display_sorted_filtered_data(url) {
  download_data(url)

    .then(lambda (downloaded_data) { filter_data(downloaded_data); } )

    .then(lambda (filtered_data)   { sort_data(filtered_data);     } )

                                   ui(sorted_data);      } )
```

This says: Add this lambda function (which is a ca... to the run queue...

This says: Add this lambda function to the run queue...

This chunk of code defines a pipeline of operations.

E ... ts work, produces a result, ... t lambda in the pipeline.



This syntax hides what's really going on – it's really just using callbacks like before!

P Syntax for ons

Whe s, it doesn't run sy ently!

This chunk of code defines a pipeline of operations.

Execution of the pipeline happens concurrently!

```
function display_sorted_filtered_data(url) {
  download_data(url)
    .then(lambda (downloaded_data) { filter_data(downloaded_data); } )
    .then(lambda (filtered_data)   { sort_data(filtered_data);     } )
    .then(lambda (sorted_data)     { update_ui(sorted_data);       } )
}

do_first_thing()
display_sorted_f
while (!done) {
  do_other_stuff();
  do_even_more_stuff();
}
```

on

This pipeline won't run to completion here!
It's just started here, and runs in the background.

```
function download_data(url){
 while (!done) {
   get_data()
   save_data()
 }
 return result;
}
```

```
function filter_data(data) {
 while (!done) {
   foo();
   bar();
 }
 return result;
}
```

```
function sort_data(data) {
 while (!done) {
   divide();
   conquer();
 }
 return result;
}
```

```
function update_ui(data) {
  ...
}
```

There's a lot of detail that we're skipping, but the important thing is to be able to recognize "Promise" syntax and know what's going on.

Promises and Errors

If this step of the pipeline generates an error...

OK, how do we deal with errors that

lo so, we use an "exception-like" syntax -

If this step of the pipeline generates an error...

```
function display_sorted_filtered_data(url) {
  download_data(url)
    .then(lambda (downloaded_data) { filter_data(downloaded_data);   } )
    .then(lambda (filtered_data)   { sort_data(filtered_data);       } )
    .then(lambda (sorted_data)     { update_ui(sorted_data);         } )
    .catch(lambda (error)          { add_to_all_errors(error);       } )
}
```

This cat

This catch handler runs.

f you don't catch

errors from any

Adding this lambda onto the

Adding this lambda onto the run queue.

Although this provides less error-handling granularity, so be careful.

Promises In a Few Different Languages

```
// JavaScript promise example
function create_and_run_promise(input) {
  run_first_computation(input)
    .then(fur                    _second_computation(output1);  })
    .then(fur                    _third_computation(output2);   })
    .then(fur                    g("Fir
```

```
  try {
    Comple      eFuture<String> result =
      Comple    bleFuture
        .supplyAsync(        () -> { return run_first_computation(input);    })
        .thenApplyAsync(output1 -> { return run_second_computation(output1); })
        .then     pplyAsync(output2 -> { return run_third_computation(output2);  });

        t.println("Final output: " + result.get());
```

Our JavaScript example looks much like our earlier pseudocode.

Once we get the final result of our promise, it'll be a String.

We start by supplying an asynchronous function to run.

Java use different terminology – CompletableFuture would be the equivalent of a Promise in our earlier examples.

And then supply other fu

This call will suspend execution of our function until we get the final result from our pipeline.

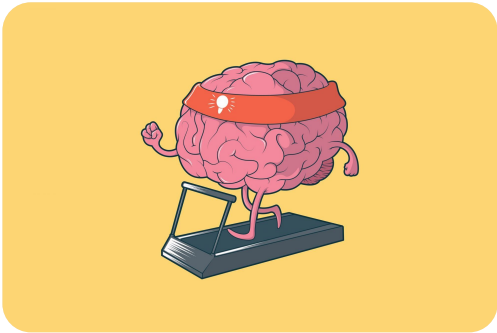
These will start running in the background immediately.

There's a Lot More To Promises

We've just scratched the surface of Promises!

For example, we didn't learn how to create a function that creates a new Promise and keeps it!

You'll only be responsible for what we learned in class on the final, but if you'd like to learn more:



https://github.com/TimothyGu/learning-promises/blob/main/by-building/README.md

ntax for Chaining Operations

et another approach with even simpler syntax.

Here' c/await version of our earlier pipeline in JavaScript:

async means that the function will run asynchronously and deliver its result at some unknown point in the future.

```
async function download_data(url) { ... }
async function filter_data(data) { ... }
async function sort_data(data) { ... }
function update_ui(data) { ... }

ync function display_sorted_filtered_data(url) {
  data = await download_data(url);      //
  ata = await filter_data(data);        //
  ta = await sort_data(fdata);          //
  update_ui(sdata);                     //
}
```

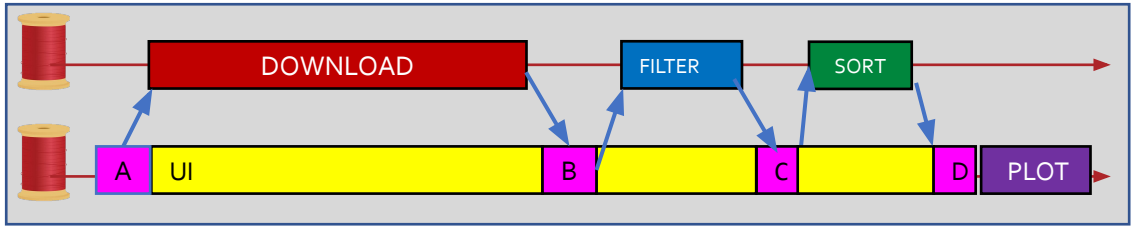
And wake me up later

And wake me up later once we get a result.

Then launch this operation in the background...

Then launch this operation in the background...

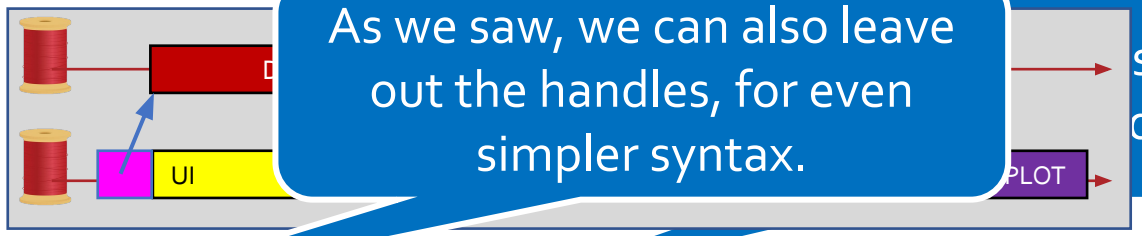
Then finally finish.



Async/Await Syntax for Chaining Operations

Let's dive a little deeper into async/await.

When you use this paradigm, there are two phases for executing tasks



sk(s)
d.

As we saw, we can also leave out the handles, for even simpler syntax.

Launch the Async Function

You request execution of a function in the background and obtain a handle to the pending task.

```
async function display_sorted_filtered_data(url) {
    data = await download_data(url);
    fdata = await filter_data(data);
    sdata = await sort_data(fdata);
    update_ui(sdata);
```

The async function immediately returns a "ha
that can be used to track its progress

The handle is NOT the function's retur
object we can use to get the return va
function eventually finishes ru

If the async task returned a result, we can get it here when the awaiting is done.

After launch, the current
ion continues running

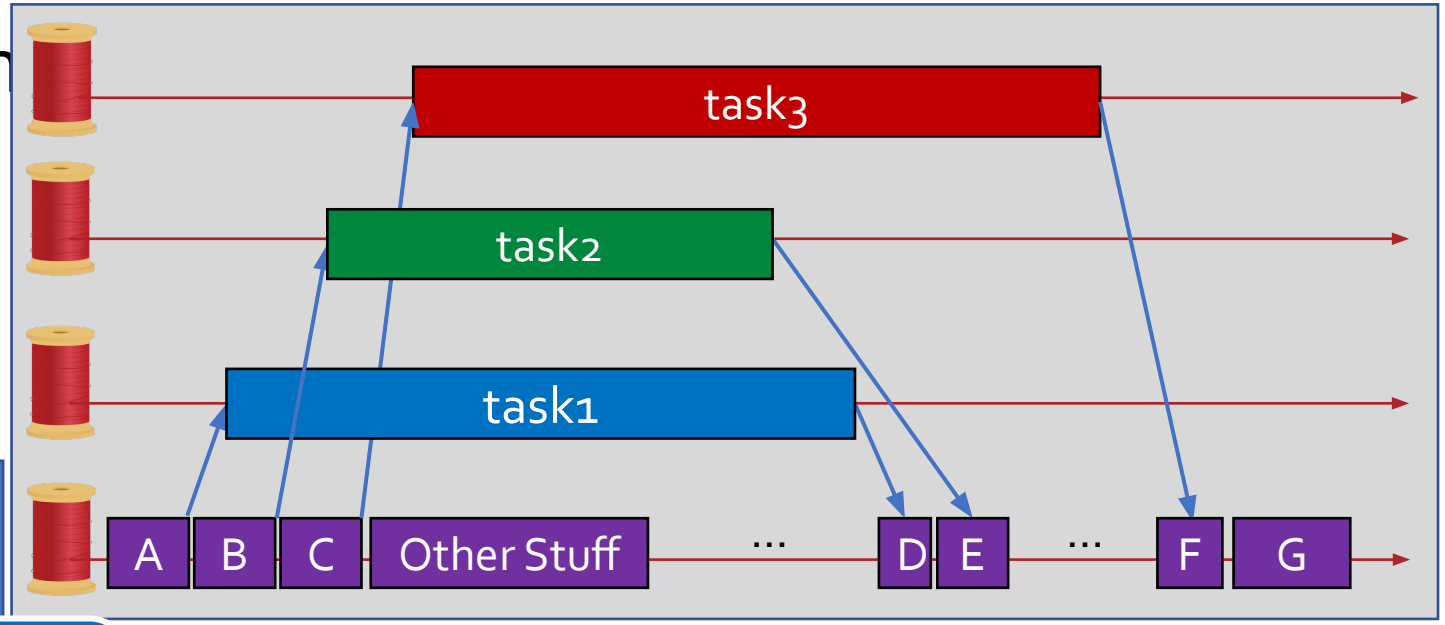
Await suspends execution of our current function until the async task finishes (and returns a result).

The code behaves in a synchronous manner, but in fact it's asynchronous!

Async/Await Syn

Of course, there's no reason you must immediately await tasks like in our previous example.

Let's see.



Launch the Async Function

You request execut
background and
penc

Wait for Completion

You may optionally wait for your launched task to complete its execution, and then

```
ync function run
h1 = task1(data1
h2 = task2(dat
h3 = task
(data3);
... // Do other stuff in the meanwhile...
result1 =  it h1;                    // D
result2 = a  2;                      // E
result3                              // F
use_res                              // G
}
```

Now we wait for our three tasks to complete!

Execution proceeds immediately to the next statements...

This immediately launches all three tasks in the background without waiting for them to finish.

Once all three finish, we can use their results.

Async/Await Implementation

Async functions are paused and can later resume execution with all their local variables and other state intact.

What technology do you think is used to implement async functions

Answer: Asyr fundamental technology a

Await is basically implemented using a type of "yield"

Our async function secretly knows where to return when it's done.

```
async def do_many_things(a):
  b = await do_thi
  c = await do_thi
  d = await do_thi
  return d
```

```
async def do_thing1(a):
  result = ... # some slow task

  return result
```

When it finishes running, it can re-activate the calling function's closure and send over the computed result!

```
async def do_many_things(a):
  b = yield_to do_thing1(a, this_closure())
  c = yield_to do_thing2(b, this_closure())
  d = yield_to do_thing3(c, this_closure())
  return d
```

```
async
esu
  yiel
re):
```

yield_to secretly passes the function's closure as a parameter to the called function, then goes to sleep waiting for a result.

This technology underlying generators and async functions is called a "coroutine".

ent Languages: JavaScript

In JavaScript we define async functions using the async keyword.

```
// JavaScr async/await
async function doesnt_return_value()
{
  // async code that doesn't
  // need to return a result
}

n does_return_value() {
e that produces a resu
_result;
}

async fu tion exa
  doesnt_ eturn_va
  h = does_returr
  r = await h;
}

example();
other_stuff();
```

Until the result is available...

And only then does it continue executing...

Blocks the function...

Calling an async function ...matically launches it in the background.

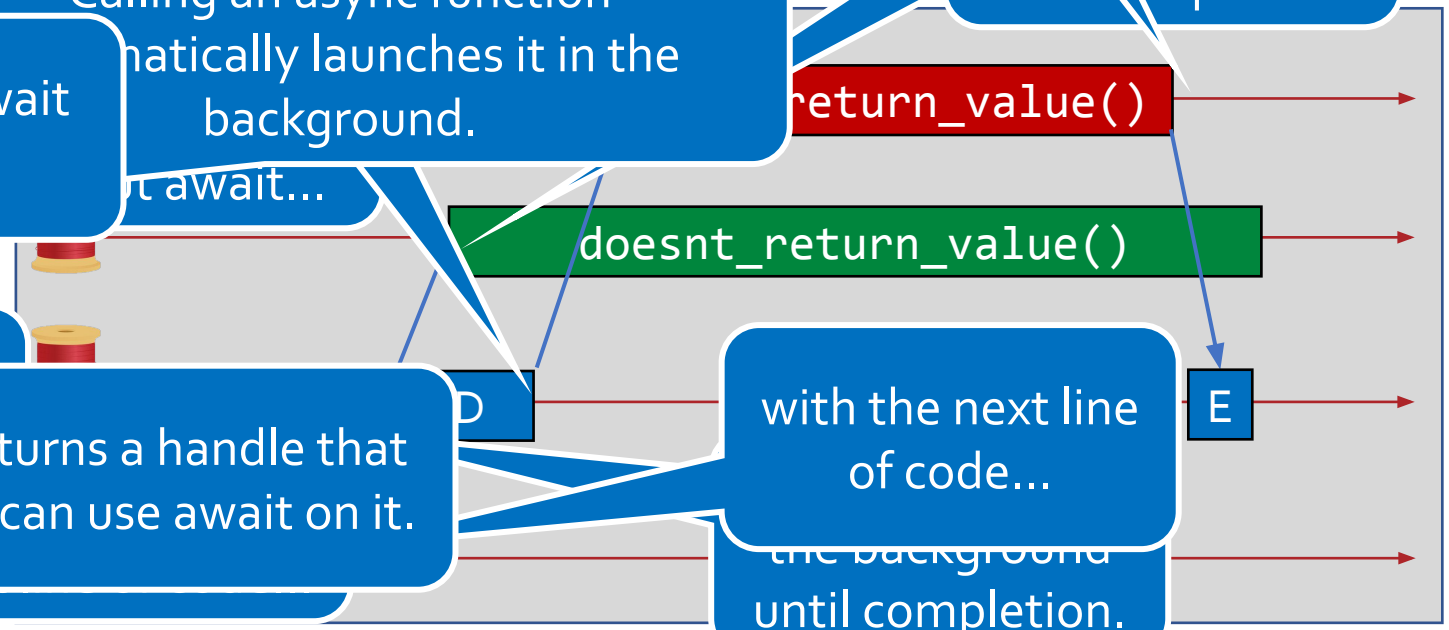
However, calling await on a handle...

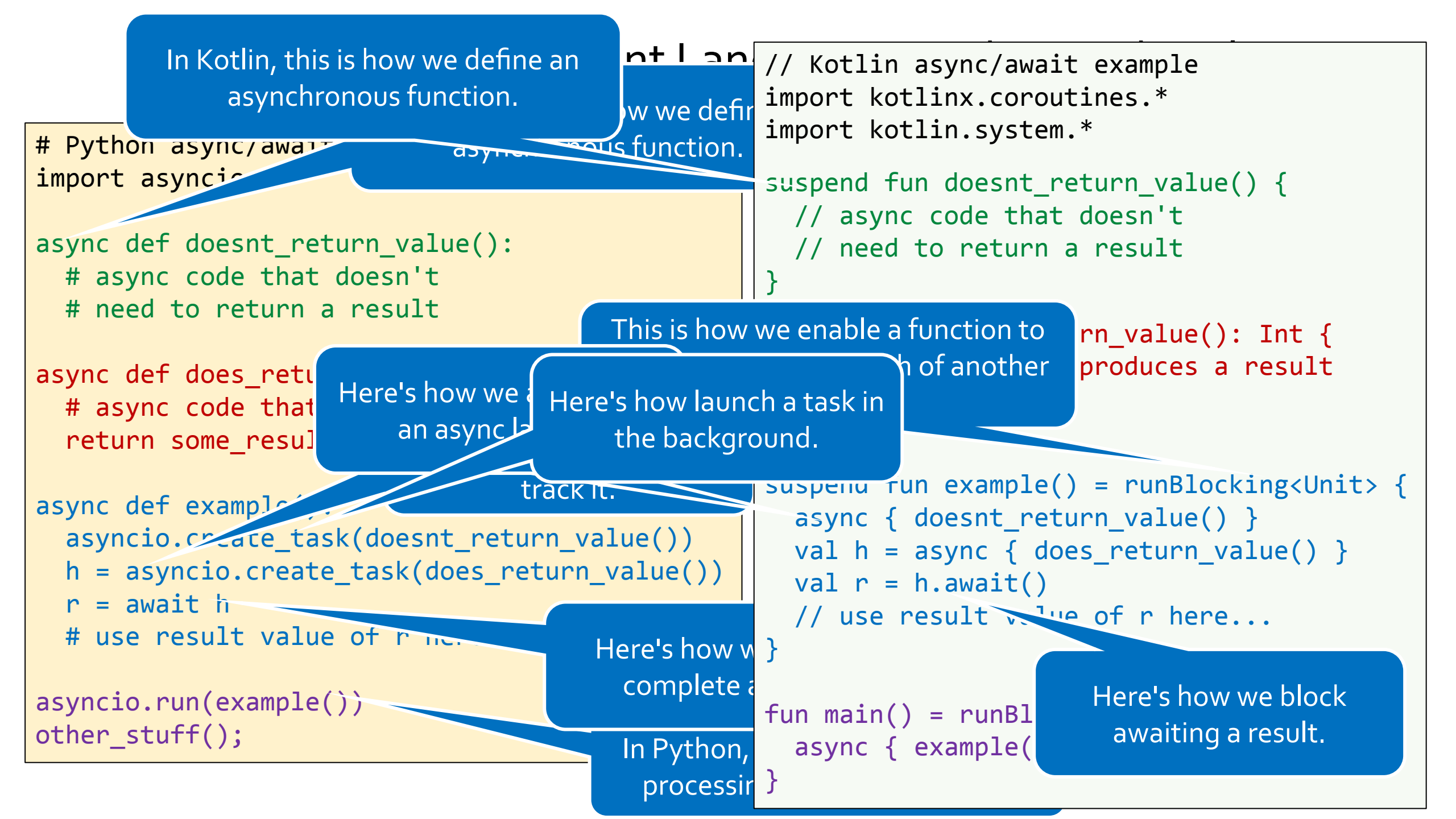
...immediately continues...

Every async function returns a handle that tracks its progress – we can use await on it.

with the next line of code...

...until completion.




In Kotlin, this is how we define an asynchronous function.
# Python async/await
import asyncio
async def doesnt_return_value():
async code that doesn't
need to return a result
async def does_return_value():
async code that
return some_result
async def example():
asyncio.create_task(doesnt_return_value())
h = asyncio.create_task(does_return_value())
r = await h
# use result value of r here...
asyncio.run(example())
other_stuff();
// Kotlin async/await example
import kotlinx.coroutines.*
import kotlin.system.*
suspend fun doesnt_return_value() {
// async code that doesn't
// need to return a result
}
rn_value(): Int {
produces a result
suspend fun example() = runBlocking<Unit> {
async { doesnt_return_value() }
val h = async { does_return_value() }
val r = h.await()
// use result value of r here...
}
fun main() = runBl
async { example(
}
This is how we enable a function to
of another
Here's how we
an async
track it.
Here's how launch a task in the background.
Here's how w
complete
In Python,
processi
Here's how we block awaiting a result.

Classif… …ncurrency

```
func foo(output_channel chan byte)
  var c byte
  for c = 'a'; c <= 'y'; c++ {
    fmt.Printf("%c", c)
    time.Sleep(100)     // millis
  }

  output_channel <- 'z'
}

func main() {
  output_channel := make(chan byte)
  go foo(output_channel)

  for i := 0; i
    fmt.Printf("%
    time.Sleep(10
  }
  fmt.Printf("\nWaiting...\n")
  fmt.Printf("\nOutput: %c\n", <-output_channel)
}
```

The two functions run concurrently with each other.

The called function can communicate back with the caller over a "channel" (aka queue).

This launches foo() so it runs concurrently.

This is the way to extract a value passed over the channel.

…e program on the left produces the following output:

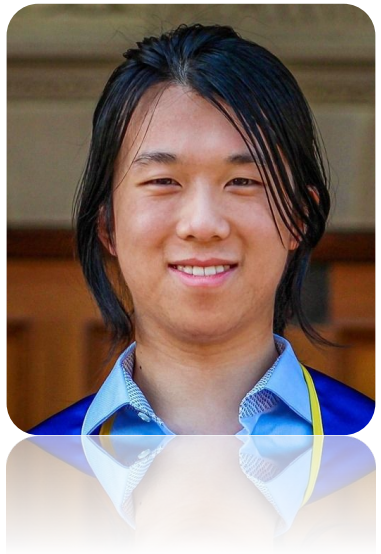
```
0ab12cd34e5fg67hi8j9
…ting...
…opqrstuvwxy
…ut: z
```

What can we conclude about concurrency in this language?

This is Go!

Want to Learn More about Async/Await?

See the excellent writeup below by our own Timothy Gu!



https://github.com/TimothyGu/learning-promises/tree/main/async-await?fbclid=IwAR0IDCnd5xGHGt1yXIlR5p5flfmLfOczpAo6wgGB4ntYakI7u8-QDCmsdFo

That's It!

Hopefully, you now have a good overview of how expression evaluation, flow control and concurrency work across programming languages!

And now... onto Prolog!

Slide Graveyard/Appendix

Callbacks, Promises and Async/Await

We've just seen three different methods (callbacks, promises, async/await) for creating pipelines of asynchronous operations.

Which approach do you like best? Why?

Async/Await

```
 function defrost() { ... }
 ...
```

```
function make_burger(frozen_patty, bun) {
  h1 = launch_background defrost(frozen_patty);
  h2 = launch_background preheat_griddle();

  defrosted_patty = wait_for_completion_and_get_result(h1)
  wait_for_completion(h2)

  h3 = launch_background fry_patty(defrosted_patty);
  h4 = launch_background toast(bun);
  fried_patty = wait_for_completion_and_get_result(h3)
  toasted_bun = wait_for_completion_and_get_result(h4)

  assemble_burger(fried_patty, toasted_bun);
}
```

```
 let's introduce a new primitive that lets a function run in the background:
 once we launch it, it's fire and forget
```

```
let's introduce a second primitive that lets us wait for a function to complete
```

```
how long will it take now?
```

Async/Await

```
  function defrost() { ... }
  ...

 function make_burger(frozen_patty, bun) {
   h1 = launch_background defrost(frozen_patty);
   h2 = launch_background preheat_griddle();

   defrosted_patty = wait_for_completion_and_get_result(h1)
   wait_for_completion(h2)

   h3 = launch_background fry_patty(defrosted_patty);
   h4 = launch_background toast(bun);
   fried_patty = wait_for_completion_and_get_result(h3)
   toasted_bun = wait_for_completion_and_get_result(h4)

   assemble_burger(fried_patty, toasted_bun);
 }
```

Ok, what if we want to do even more in parallel

```
function make_fries() {   ... } // 10m
```

let's trace through it

```
function make_meal() {
 frozen_patty = Patty();
 bun = Bun();
 frozen_fries = Fries();

 h1 = launch_backround make_burger(frozen_patty, bun);
 h2 = launch_background make_fries(frozen_fries);

 burger = wait_for_completion_and_get_result(h1)
 fries = wait_for_completion_and_get_result(h2)
}
```

Async/Await

```
function make_burger(frozen_patty, bun) {
  defrosted_patty = defrost(frozen_patty);          // 30m  (returns a "patty")
  preheat_griddle();           // 10m  (doesn't return value)
  fried_patty = fry_patty(defrosted_patty);              // 5m
  toasted_bun = toast(bun);              // 5m
  assemble_burger(fried_patty, toasted_bun);            // 3m
}
```

```
const washVeggies = () => {
  console.log('Start washing!');
  const start = Date.now();         // Get a reference of the
current time.
  while (Date.now() < start + 2000) {} // continuosly checks
2000 milliseconds has passed.
  console.log('Veggies washed!');
};
```

```
async function make_burger(frozen_patty, bun) {
  task1 = defrost_patty(frozen_patty);          // 30m  (returns a "patty")
  task2 = preheat_griddle();         // 10m  (doesn't return value)
  defrosted_patty = await task1;
  await task2;

  task3 = fry_patty(defrosted_patty);              // 5m
  task4 = toast(bun);              // 5m
  fried_patty = await task3;
  toasted_bun = await task4;
  burger = assemble_burger(fried_patty, toasted_bun);            // 3m
  return burger;
}
```

what are the rules when something runs in the background
what are the rules when something blocks

Async/Await

```
const washVeggies = () => {
   console.log('Start washing!');
   const start = Date.now();          // Get a reference of the
current time.
   while (Date.now() < start + 2000) {} // continuosly checks
2000 milliseconds has passed.
   console.log('Veggies washed!');
};
```

what are the rules when something runs in the background
what are the rules when something blocks

Callbacks In (One) Other Language

```
// Swift
func createOurButton() {

  let myFirstButton = UIButton()
  myFirstButton.setTitle("Click Me", for: .normal)
  ...
  myFirstButton.addTarget(self,
                          action: #selector(pressed),
                          for: .touchUpInside)
}

@objc func pressed() {
  var alertView = UIAlertView()
  alertView.addButtonWithTitle("Ok")
  alertView.title = "Get Excited!"
  alertView.message = "The callback was called!"
  alertView.show()
}
```

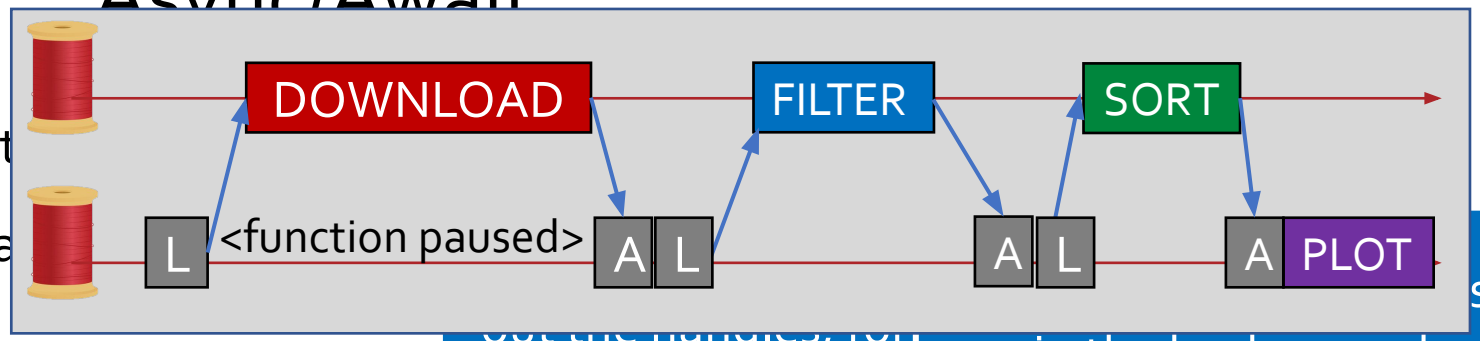
Here's how we specify our callback function.

It's called if we click and release on the button.

Async/Await

The Async/Await model is yet anot

When you use this pa



Launch the Async Function

You request execution of one or more functions in the background and obtain handle(s) to the pending task(s).

Wait for Completion

You may optionally wait for your task(s) to complete their execution, and then obtain their result.

```
function display_sorted_filtered_data(url) {
  data = await launch_async download_data(url);
  fdata = await launch_async filter_data(data);
  sdata = await launch_async sort_data(fdata);
  update_ui(sdata);
}
```

even simpler syntax.

in the background.

After launch, the current function continues running

This provides the same semantics as our promise-based code!

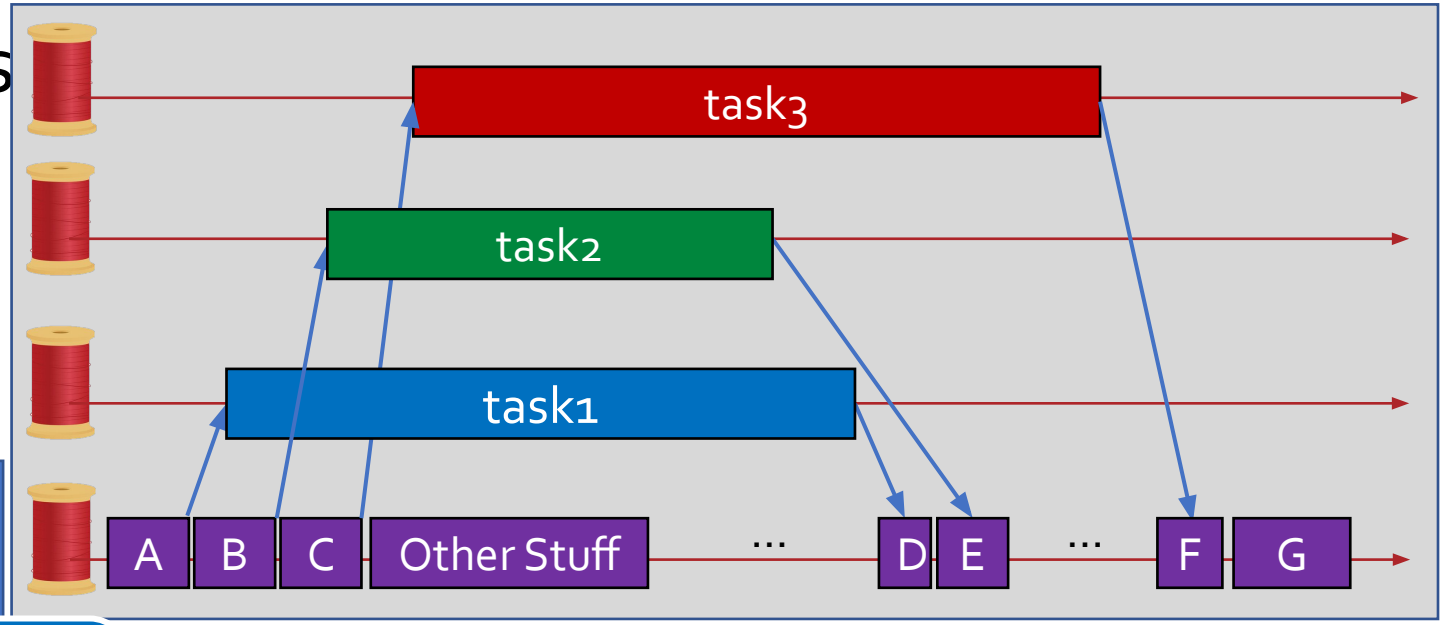
Once we get a result, the thread continues executing.

The code looks synchronous, but in fact it's asynchronous!

As

Of course, there's no reason you must immediately await tasks like in our previous example.

Let's see.



Launch the Async Function

You request exec
functions in the b
handle(s) to th

Wait for Completion

You may optionally wait for your task(s) to complete their execution, and then obtain

```
ync function run
  h1 = launch_asyn
  h2 = launch_a
  h3 = lau  _async
  ... // Do other stuff in the meanwhile...
  result1 =  it h1;                    // D
  result2 = a  2;                      // E
  result3                              // F
  use_res                              // G
}
```

Now we wait for our three tasks to complete!

Execution proceeds immediately to the next statements...

This immediately launches all three tasks in the background without waiting for them to finish.

Once all three finish, we can use their results.

Async/Await

Most languages also allow you to wait on multiple handles at the same time before proceeding, almost like a fork-join.

Launch the Async Function

You request execution of one or more functions in the background and obtain handle(s) to the pending task(s).

Wait for Completion

You may optionally wait for your task(s) to complete their execution, and then obtain their result.

```
async function run(data1, data2, data3) {
  h1 = launch_async task1(data1);
  h2 = launch_async task2(data2);
  h3 = launch_async task3(data3);

  ... // Do other stuff in the meanwhile...

  await_all [h1, h2, h3];

  ... // Continue here once task1-task3 finish
}
```

ent Languages: JavaScript

In JavaScript we define async functions using the async keyword.

```
// JavaScr async/await
async function doesnt_return_value()
{
  // async code that doesn't
  // need to return a result
}

n does_return_value() {
e that produces a resu
_result;
}

async fu tion exa
  doesnt_ eturn_va
  h = does_retur
  r = await h;
}

example();
other_stuff();
```

Until the result is available...

And only then does it continue executing...

Blocks the function...

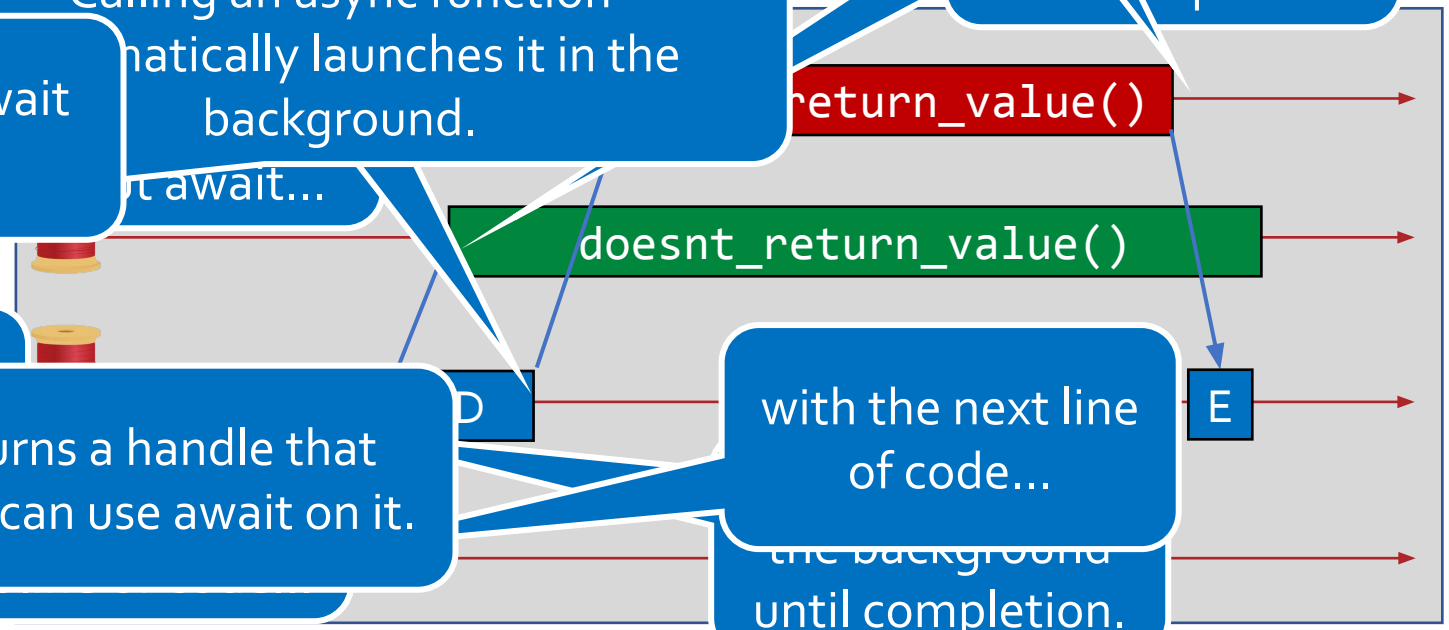
Calling an async function ...matically launches it in the background.

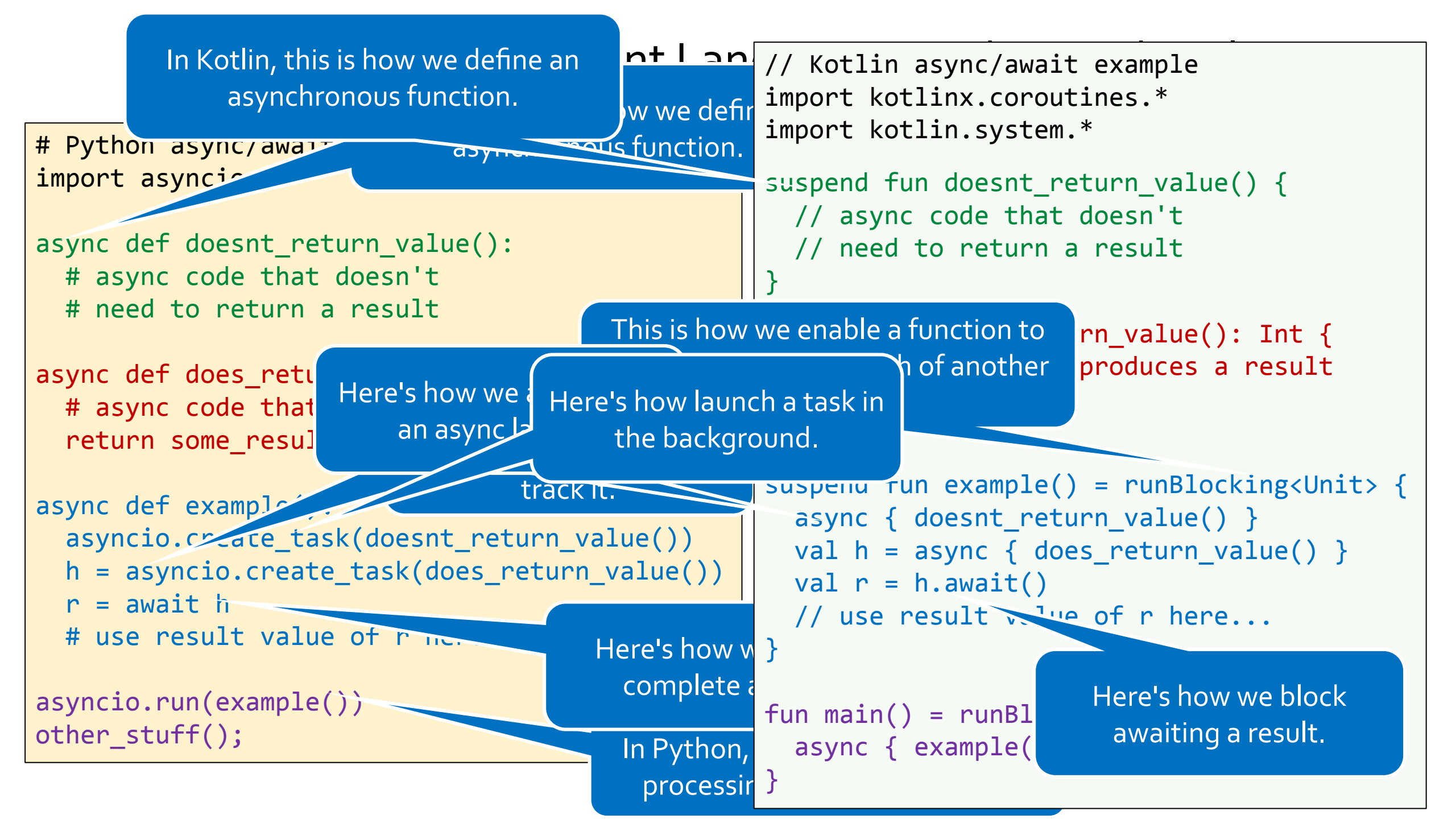
However, calling await on a handle...

...immediately continues...

An async function returns a handle that tracks its progress – we can use await on it.

with the next line of code...




In Kotlin, this is how we define an asynchronous function.
# Python async/await
import asyncio
async def doesnt_return_value():
async code that doesn't
need to return a result
async def does_return_value():
async code that
return some_result
async def example():
asyncio.create_task(doesnt_return_value())
h = asyncio.create_task(does_return_value())
r = await h
# use result value of r here...
asyncio.run(example())
other_stuff();
// Kotlin async/await example
import kotlinx.coroutines.*
import kotlin.system.*
suspend fun doesnt_return_value() {
// async code that doesn't
// need to return a result
}
rn_value(): Int {
produces a result
suspend fun example() = runBlocking<Unit> {
async { doesnt_return_value() }
val h = async { does_return_value() }
val r = h.await()
// use result value of r here...
}
fun main() = runBl
async { example(
}
This is how we enable a function to
of another
Here's how we
an async
track it.
Here's how launch a task in the background.
Here's how w
complete
In Python,
processi
Here's how we block awaiting a result.

Languages have evolved to make this easier

Fork/join (C#, ??)
- used for CPU-bound tasks

Async + callback (javascript, ???): used for UI (app, page), IO
- button example (javascript), create thread to DL page, then call callback (python) – for UI events, event completion
- IO downloading a page
- event loop: queue them up, schedule on a single thread, block on IO goes to

Fork-join for IO: Async/await (goroutines, async/await javascript): used for IO
- queue of async functions: queue them up, schedule on a single thread, block on IO goes to next thing on the event loop - callbacks

Promises/Futures
- Javascript example, swift ???
- https://sodocumentation.net/cplusplus/topic/9840/futures-and-promises?fbclid=IwAR0BPNvbZnUrX42djSmgyA8QybA1Y9bnsHsXXhs7e1H6JEUVL4-Nf3eU1cw

Coroutines (python)

```
# python async/await example
import asyncio

async def asyncfunction():
    print('Hello')
    await asyncio.sleep(10)
    print('World')
    return 10
```

Callback Challenges

Todo

Promise Version of Our Downloader

Create a new promise and provide it with a lambda function to run
The lambda function launches an async task of:
   resolve(do_initial_step())
The promise is now pending and the current function continues running
We can call a .then() method on a promise which will only run when the initial promise is resolved
The then method takes in a lambda function which is passed the resolved value of the previous promise
The then method may then create a new promise and return it
We can then have another then method call, which uses this new promise

```
function download_data() {
  return "data";
}

function filter_data(data) {
 return "filter(" + data + ")";
}

function sort_data(data) {
 return "sort(" + data + ")";
}

function display_data(data) {
  console.log(data);
}

function download_filter_sort_show() {
  let promise = new Promise((resolve, reject) => {
      setTimeout(() => { resolve(download_data()); }, 0);
    });

  promise.then(function (data) {
    new_prom = new Promise((resolve, reject) => {
      setTimeout(() => { resolve(filter_data(data)); }, 0);
    });
    return new_prom;
  })
  .then(function (data) {
    new_prom = new Promise((resolve, reject) => {
      setTimeout(() => { resolve(sort_data(data)); }, 0);
    });
    return new_prom;
  })
  .then(function (data) {
    display_data(data);
  })
  console.log('foo');
}

download_filter_sort_show();
  console.log('bar');
```

initial promise lambda runs synchronously
all other then()s are queued and run after the call stack is exhausted
returning a value from a then() wraps it in a resolved promise as well

Coroutines, Continuations and Generators

```
# Generator example

def countto(n: int):
   print('foo')
   i=0
   while i < n:
      yield i
      i += 1
 # code reaches here or a return and that ends next


# Example use: printing out the integers from 10 to 20.
# Note that this iteration terminates normally, despite
# countfrom() being written as an infinite loop.

for i in countfrom(10):
   if i <= 20:
      print(i)
   else:
      break

or:

gen = countfrom(10)   #  gen is a python object – the function
isn't run yet
 # the first call to next runs the function for the first time until
yield, stores the program counter & closure, then returns the
value
 next(gen) #returns 10
# 2nd and beyond calls continues the function just after the
last yield until the next yield, stores the pc & closure, returns
the value
```

Coroutine is one of several procedures that take turns doing their job and then pause to give control to the other coroutines in the group. (C#, Python, Ruby, PHP, JavaScript, Rust, …)

- http://www.dabeaz.com/coroutines/Coroutines.pdf
    - Good comparison to using objects for the same result
- Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.
- In the case of threads, it's an operating system (or run time environment) that switches between threads according to the scheduler. While in the case of a coroutine, it's the programmer and programming language which decides when to switch coroutines. Coroutines work cooperatively multitask by suspending and resuming at set points by the programmer.
- Subroutines are special cases of coroutines. When subroutines are invoked, execution begins at the start, and once a subroutine exits, it is finished; an instance of a subroutine only returns once, and does not hold state between invocations.
- See: https://www.geeksforgeeks.org/coroutine-in-python/

Continuation is a "pointer to a function" you pass to some procedure, to be executed ("continued with") when that procedure is done.

Classify That Language: Concurrency

```
void Page_Load(Object sender, EventArgs e) {
  Button1.Click += new EventHandler(this.handler1);
  Button2.Click += new EventHandler(this.handler2);
}

async void handler1(Object sender, EventArgs e) {
  var result = await DoQuickOperation();
  DoSomethingWithResult1(result);
};

async void handler2(Object sender, EventArgs e) {
  var result =
    await Task.Run( () => MultTwoLarge
  DoSomethingWithResult2(result);
};
```

So while await syntax is used to wait for both operations...

and, fast operations can e event loop, since they won't noticeably delay the UI.

Task.Run() is used to launch a background thread, running independently of the event loop.

ons cannot run in the use they'd cause the UI while they run.

The code to the left runs when a web page is first loaded and sets up callbacks for button clicks.

This language uses similar syntax for both multi threading and event-loop-based concurrency.

Which button executes code via the event loop, code in a background ad?

This is C#!

Generators in other languages (PHP, Ruby, C#

```php
#php
function fibonacci()
{
    $last = 0;
    $current = 1;
    yield 1;
    while (true) {
        $current = $last + $current;
        $last = $current - $last;
        yield $current;
    }
}

foreach (fibonacci() as $number) {
    echo $number, "\n";
}
```

```
// C# Method that takes an iterable input (possibly an a
// and returns all even numbers.
public static IEnumerable<int> GetEven(IEnumerable<
numbers)
{
    foreach (int number in numbers)
    {
        if ((number % 2) == 0)
        {
            yield return number;
        }
    }
}
```

Iteration

"True" Iterators

- **Subroutine with `yield` statements**
  - Each `yield` "returns" another element
- **Popular, e.g., in Python, Ruby, and C#**
- **Used in a `for` loop**
  - Example (Python):

```
# range is a built-in iterator
for i in range(first, last, step):
    ...
```

Example: Binary Tree

```
class BinTree:
    def __init__(self, data):
        self.data = data
        self.lchild = self.rchild = None

    # other methods: insert, delete lookup, ...

    def preorder(self):
        if self.data is not None:
            yield self.data
        if self.lchild is not None:
            for d in self.lchild.preorder():
                yield d
        if self.rchild is not None:
            for d in self.rchild.preorder():
                yield d
```

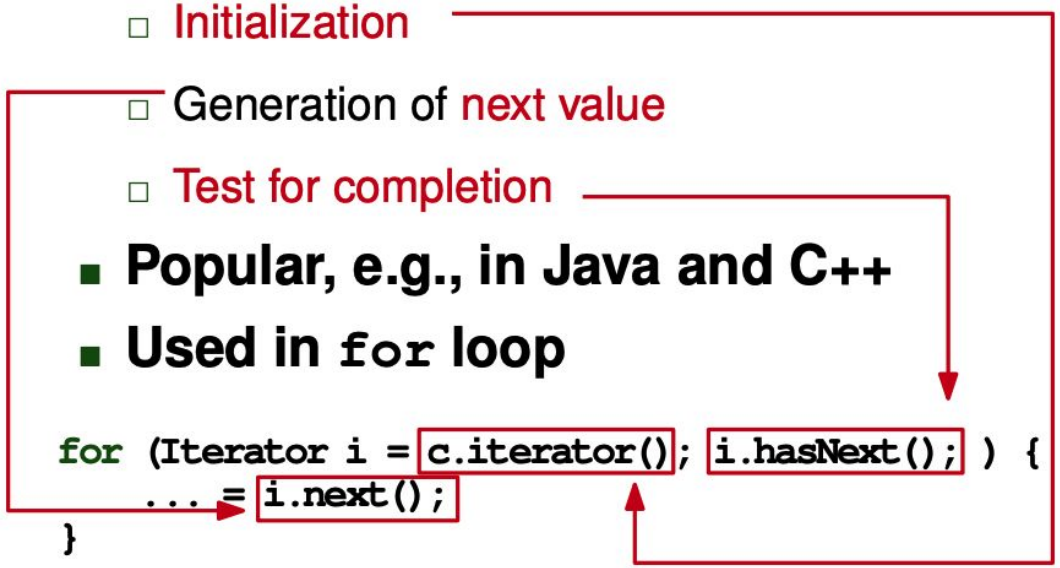
Iterator Objects

- **Regular object with methods for**
  - Initialization
  - Generation of next value
  - Test for completion
- **Popular, e.g., in Java and C++**
- **Used in `for` loop**

```
for (Iterator i = c.iterator(); i.hasNext(); ) {
    ... = i.next();
}
```

```
for (Element e : c) {
    ...
}
```


- **Regular object with methods for**
  - Initialization
  - Generation of next value
  - Test for completion
- **Popular, e.g., in Java and C++**
- **Used in `for` loop**



Classify That Language: Iteration

Consider the following program…

TODO

This is ???!

Iterating with First-Class Functions

- **Two functions**
  - One function about what to do for each element
  - Another function that calls the first function for each element
- **Example (Scheme):**

```
(define uptoby
  (lambda (low high step f)
    (if (<= low high)
      (begin
        (f low)
        (uptoby (+ low step) high step f))
      '())))
```

Iterating with First-Class Functions (2)

- **Originally, proposed in functional languages**
- **Nowadays, available in many modern PLs through libraries**
  - E.g., Java

```
mySet.stream().filter(e -> e.someProp > 5)
```

  - E.g., JavaScript

```
myArray.filter(e => e.someProp > 5)
```

Continuations

In low-level terms, a continuation consists of a code address and a referencing environment to be restored when jumping to that address. In higher-level terms, a continuation is an abstraction that captures a context in which execution might continue. Continuations are fundamental to denotational semantics. They also appear as first-class values in certain languages (notably Scheme and Ruby), allowing the programmer to define new control- flow constructs.

> The implementation of continuations in Scheme and Ruby is surprisingly straightforward. Because local variables have unlimited extent in both lan- guages, activation records must in general be allocated on the heap. As a re- sult, explicit deallocation is neither required nor appropriate when jumping through a continuation; frames that are no longer accessible will eventually be reclaimed by a general purpose garbage collector (to be discussed in Sec- tion 7.7.3). Restoration of state (e.g., saved registers) from escaped routines is not required either: the continuation closure holds everything required to resume the captured context.

Multiprocessing

Cooperative multitasking, co-routines

9.13 Coroutines 407

- A coroutine is a subprogram that has **multiple** entries and co
  directly in Lua
- The coroutine control mechanism is often called the symme
  coroutines are more equitable
- It also has the means to maintain their status between activation
- This means that coroutines must be **history sensitive** and thus h
- Secondary executions of a coroutine often begin at points other
- The invocation of a coroutine is named a **resume** rather than a
- The first resume of a coroutine is to its beginning, but subsequ
  after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide quasi-concurrent execution of program
  execution is interleaved, but not overlapped

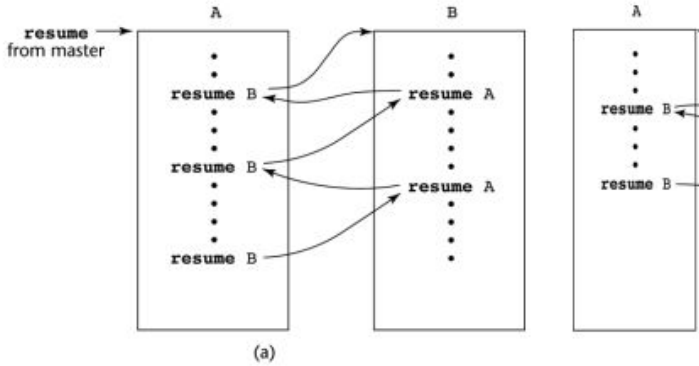
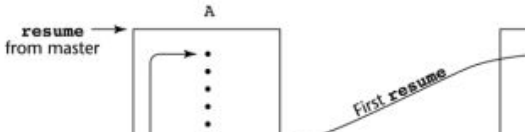


Figure 9.3 Two possible execution control sequences for two



Lazy Evaluation/Short Circuiting

5.1.1 Lazy evaluation
Evaluating expressions with boolean operators (and, or, not) is another area where the value of an expression
is not well-defined without a semantic definition. Consider the expression below.

```
if( a || (b && c && d) )
/* do something */;
```

The values a, b, c, and d may all be functions, possibly even functions that require lots of resources to

compute. Note that, if a is true, then the rest of the expression is irrelevant to the truth of the entire state-
ment. Only if a is false, does the rest of the statement require evaluation. The expression (b && c &&

d) also permits lazy evaluation, because if any of the individual items are false, the entire statement is

1. *sequencing:* Statements are to be executed (or expressions evaluated) in a certain specified order—usually the order in which they appear in the program text.
2. *selection:* Depending on some run-time condition, a *choice* is to be made among two or more statements or expressions. The most common selection constructs are `if` and `case` (`switch`) statements. Selection is also sometimes referred to as *alternation*.
3. *iteration:* A given fragment of code is to be executed repeatedly, either a certain number of times or until a certain run-time condition is true. Iteration constructs include `while`, `do`, and `repeat` loops.
4. *procedural abstraction:* A potentially complex collection of control constructs (a *subroutine*) is encapsulated in a way that allows it to be treated as a single unit, often subject to parameterization.
5. *recursion:* An expression is defined in terms of (simpler versions of) itself, either directly or indirectly; the computational model requires a stack on which to save information about partially evaluated instances of the expression. Recursion is usually defined by means of self-referential subroutines.
6. *concurrency:* Two or more program fragments are to be executed/evaluated "at the same time," either in parallel on separate processors or interleaved on a single processor in a way that achieves the same effect.
7. *nondeterminacy:* The ordering or choice among statements or expressions is deliberately left unspecified, implying that any alternative will lead to correct results. Some languages require the choice to be random, or fair, in some formal sense of the word.

- First Lecture
  - Short circuiting (sidefx)
  - Looping mechanisms, break, continue?
- Second lecture
  - generators in python/yield
  - Concurrency
  - Ownership (rust)
  - Fork/join
  - Global Interpreter Lock

- Parking lot
  - "definite assignment" compiler checking that a variable is assigned across all dynamically at runtime

Iteration: Looping With a Counter

```
// Perl
for(my $x = 1; $x <= 5; $x++) {
  for(my $y = 1; $y <= $x; $y++) {
    print "*";
  }
  print "\n";
}
```

Are the values iterators

Conditionals: Beyond If Statements

First let's discuss the conditional te

Here are its semantics: If b

```
// C++, C#, Java, JavaScript, Swift, e
beverage = age < 21 ? "Coke"  : "Beer"
```

```
se "Beer"
```

```
// Kotlin cond
```

There ar guages:

Here B

```
// C#, JavaScr
my_bev = drink
```

```
is operator
k ?: "water"
```

Finally, som ges support an inline version of this operator!

```
// C#, Kotlin, JavaScript, Swift, ...
name = article?.co_author?.name
```

```
# Ruby's slightly different syntax
name = article&.co_author&.name
```

FUN FACT

This is called the elvis operator because if you turn it on its side, it looks like Elvis!

?:

```
if (article == null)
  name = null
else if (article.co_author == null)
  name = null
else
  name = article.co_author.name
```

Classify That Language: Multi-way Selection

```
let fraction = (5, 6)
...
switch fraction {
 case let (num, denom) where denom == 0:
  print("The value is infinity")
 case let (num, denom) where num >= denom:
  print("\(num/denom) is greater than 1")
 case let (num, denom):
  print("\(num/denom) is a fraction")
}
```

Consider the following program…

TODO

This is Swift!

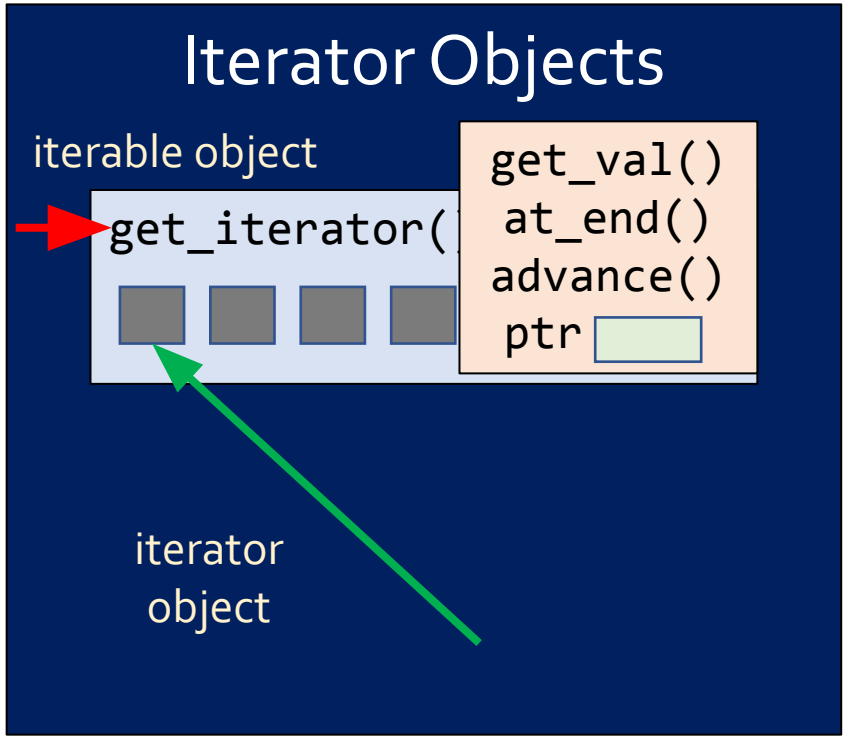
Looping and Iterators

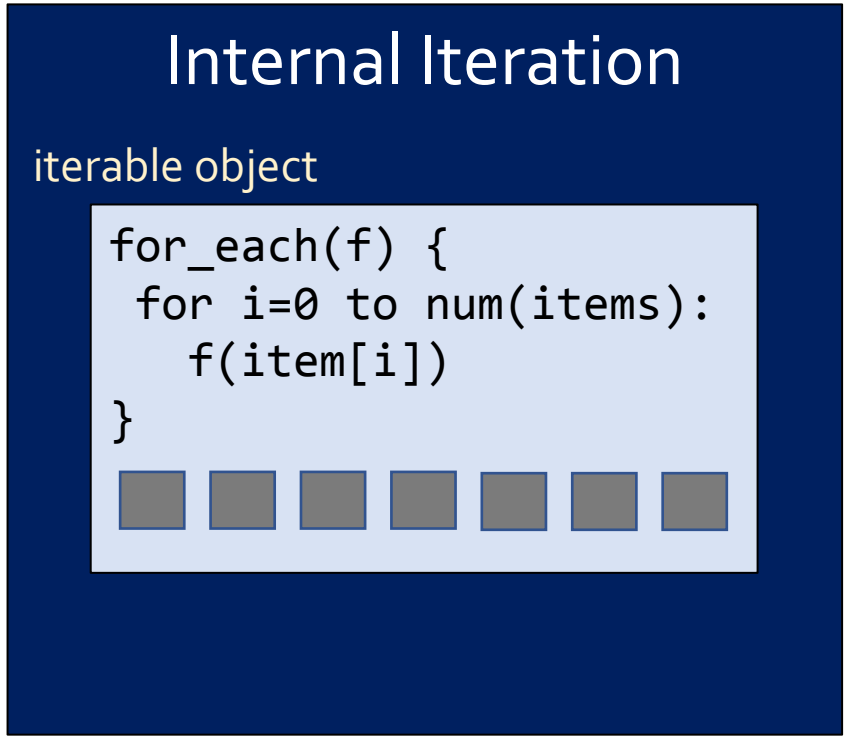
Next let's learn how iteration actually is implemented under the hood:

```
vector<string> fruits = {"plum","pear",...};
for (auto f: fruits)
  cout << f << endl;
```

```
for i in range(1,10):
  print(i)
```

There are two high-level approaches:





Iterators

What's the big picture?

An iterable class is one which contains a collection of values that can be processed sequentially.

Examples: arrays, lists, sets, maps, ranges

In OOP languages, we use an abstraction called an "iterator" to point at, retrieve, and advance through the values in an iterable object.

```
set<string> nerds;
... // add items to nerds set

set<string>::iterator it;
it = nerds.begin();
while (it != nerds.end()) {
   cout << *it << endl;
   ++it;
}
```

Concurrency Approaches

Let's break down the high-level approaches to concurrency in modern languages.

Full Multithreading A program may launch any number of threads, and their execution is interleaved across available cores	C++, Java, C#, Kotlin, Rust, etc.
Limited Multithreading The language supports concurrent execution, but there are constraints preventing some kinds of parallelism	Python
Simulated Multithreading The language supports concurrent execution, but all execution is actually limited to a single thread	JavaScript, TypeScript, etc.

```
<html><body>
<div>The current weather is:</div>
<div id="weather">No weather data</div>

<script>
function update_weather_text(txt)
 { document.getElementById("weather").innerHTML = txt; }

function update_ui_when_data_arrives()
 { update_weather_text(this.responseText); }

function download_weather() {
  var req = new XMLHttpRequest();
  req.onreadystatechange = update_ui_when_data_arrives;
  req.open("GET", "https://www.weather.com/...", true);
  req.send();  // starts execution in background
}

download_weather()
update_weather_text("Retrieving weather...")
</script>
</body></html>
```

req

onreadystatechange

responseText

"77 degrees and cloudy"

rogramming Without Concurrency

of how a
ency wou

Updating the display doesn't happen until the next iteration.

```
# Browser pse    ode
def render_bro    r_UI():
  while True:
    display_UI_in_browser()
    check_for_mouse_and_keyboard_events()
    run_next_js_statement_to_completion()
```

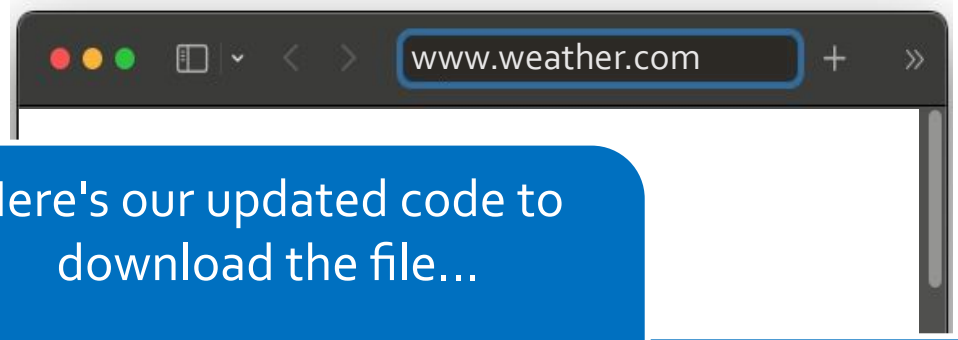
The browser hangs until the operation finishes!

This changes the contents of the page, but doesn't update the screen just yet!

This is called a "blocking" operation.

```
<div>The c      weather is:</div>
<div id="weather      weather data</div>

<script>
function update_weather(data) { ... }

data = download("www.weather.com/...");
update_weather(data);
</script>

</body></html>
```

UI Pro ncurrency and Callbacks




```
# Browser pseudocode
def render_brow_UI():
  while True:
    display_UI_in_browser()
    check_for_mouse_and_keyboard_
    run_next_js_statement_to_comp
```

Execution continues immediately in our browser main loop!

Here's our updated code to download the file...

...version that can run background...

...at we passed our ...weather() function as a parameter.

This changes the contents of the page, ...t update the ...just yet!

Our downloader will call the callback to updates the page contents with the temperature, etc.

When the file ... downloading...

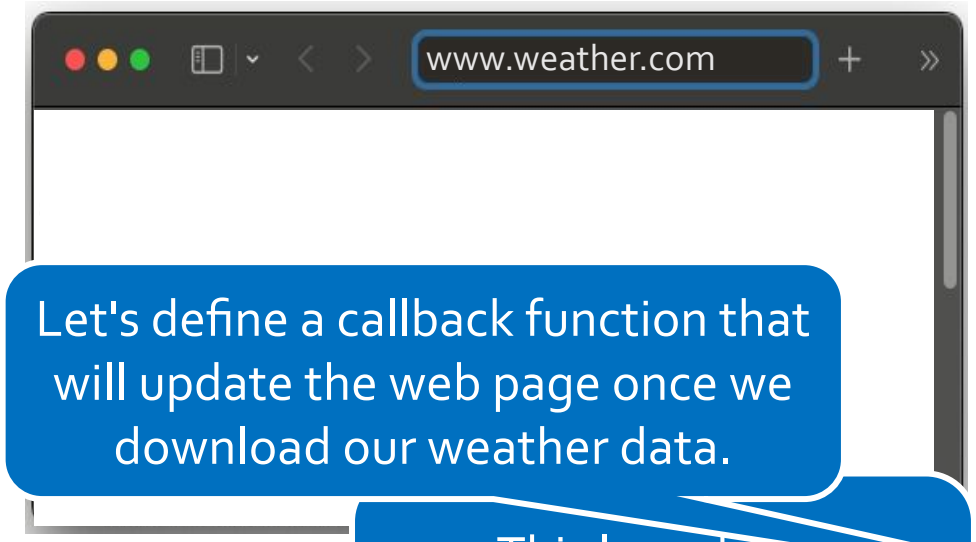
And we'll add a status update to our page so the user knows what's happening.

```
...urrent ...her is:</div>
<div id="we...er" ... weather data</...
...
<script>
function update_...eather(data) { ...
...
data = download_concur("www.weather...com/...",
                       update_weather);
update_weather("Downloading data...");
</script>
</body></html>
```

...er Object

callback

UI Programming With Concurrency and Callbacks



Let's define a callback function that will update the web page once we download our weather data.

This launches a loader in the background.

When the file finishes downloading...

And we'll add a status update to our page so the user kn what's happening.

Notice that we pass our show_weather() callback as a parameter.

Our downloader will call the callback to update the page contents with the temperature, etc.

callback

```
<html><body>
<div>The current weather is:</div>
<div id="wthr_txt">No weather data</div>

<script>

                                                    ;

</script>
</body></html>
```

Let's use a version of the downloader that can run concurrently (in the background).

Solu

The worker calls this function if it has a successful result.

The worker calls this function if it has an error.

eate a s age has two parts:

```
function worker_for_stage_x(resolve_fn, reject_fn) {
  output = do_the_actual_work_for_stage_x();
  if (was_successful(output))
                                    ge x!");
}


functi  stage_x()
  p = Promise(
        lambda (res
          run_in_background worker_for_stage_x(resolve_fn, reject_fn)
        }
      );
  return p;
}
```

This function creates a new Promise object.

The worker does it's work.

Otherwise, the worker calls the reject_fn() function.

And passes in a lambda function that calls the worker.

First, we define a worker function that does the actual work.

The worker takes two functions as its arguments.

Next, we create a separate function that creates a promise for this stage of the pipeline.

What's a Promise?

A promise is an object that launches an asynchronous task and tracks its status.

When you create a Promise object, you give it a lambda function to run.

The Promise object c'tor runs the lambda function, which is responsible for kicking off a background task.

Then the promise constructor then immediately returns, and execution resumes with the next statement.

At some later point, the task finishes (in the background) and sends its output/status back to the promise object.

At this point, the promise is fulfilled.

A promise object may have subscribers.

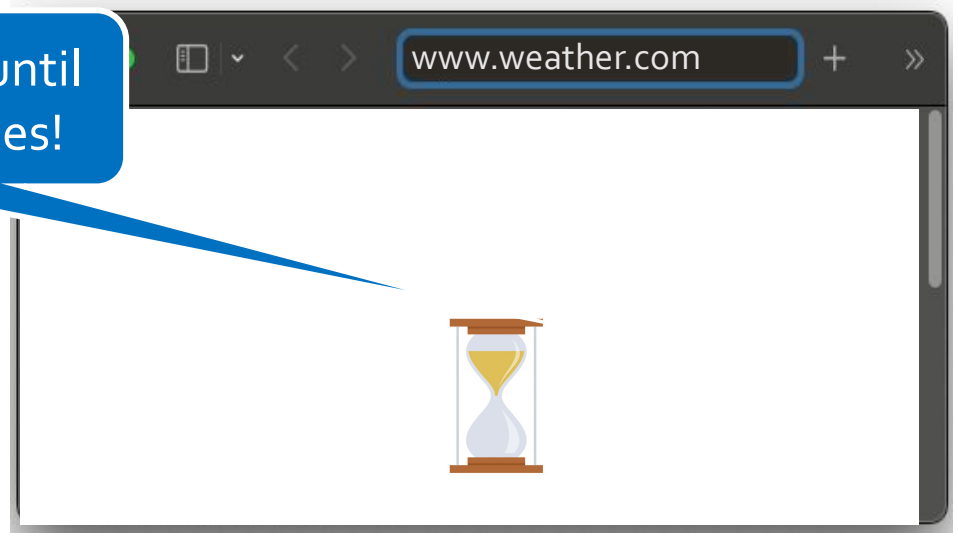
You may register a lambda function that runs once a lambda completes

For the gory details, see a great explanation from Timothy Gu in the slide notes!

UI Programming Without Concurrency 2

Here's an example of how java
doesn't use concurrency wo

The browser hangs until the operation finishes!



Clearly, if we have to run a function that takes a while to run, it'll freeze our UI.

So we have to figure out a way to run such a function in the background.

This is called a "blocking" operation.

And then get the result once it's done.

This way, the rest of the page can continue handling user interaction without stalling.

```
<html><body>
<div>The current weather is:</div>
<div id="weather">No weather data</div>

<script>
function update_html(tag,value) { ... }

download("www.weather.com/...",
   lambda (data) {
    update_html("weather", data); });
// other javascript code...
```

What's a Promise?

A Promise is an object that launches a task, tracks its execution, and receives its result once the task completes.

When you create a Promise ob… it a lambda function to run – the…

The Promise objec… function, which laun…

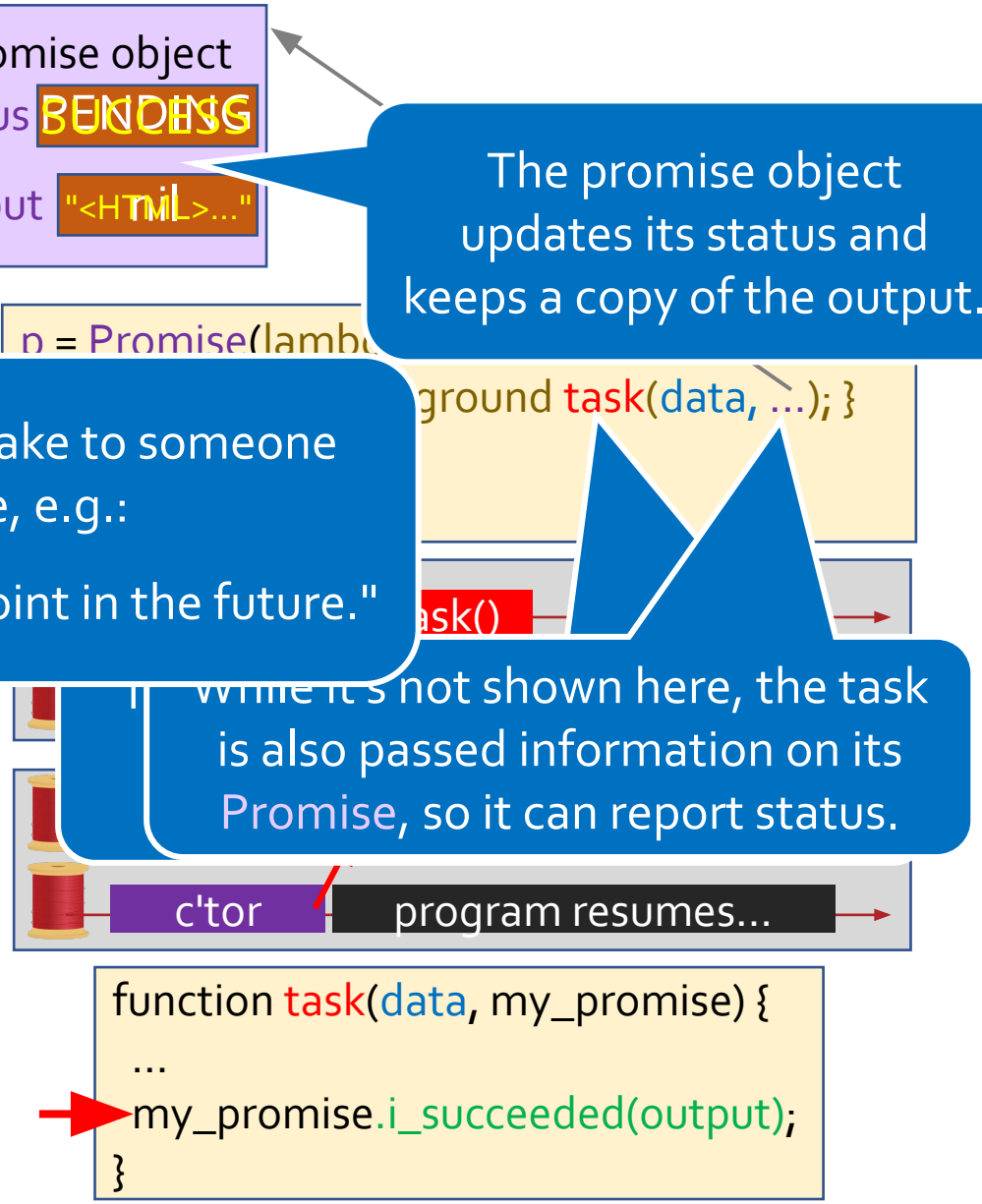
Then the Promise's constructor immediately returns, and the main program resumes normally.

At some later point, the background task finishes and sends its output/status back to its Promise object.

At this point, the Promise is "fulfilled," meaning the background task has delivered its result to the Promise.

Think of a Promise like a promise you make to someone that you'll get something done, e.g.:

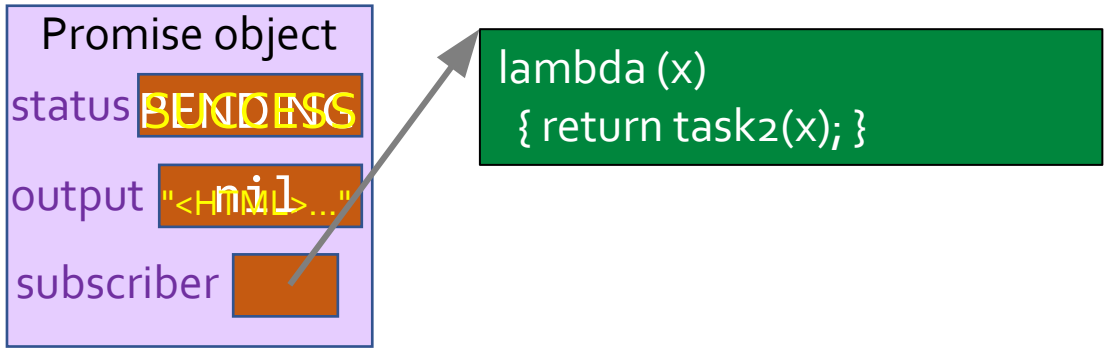
"I promise to deliver you a taco at some point in the future."



Sarcastic Robot Says:

"Many built-in language libraries auto-matically support promises, making them easy to use with simple syntax.

But wait 'til you have to write your own! :/"

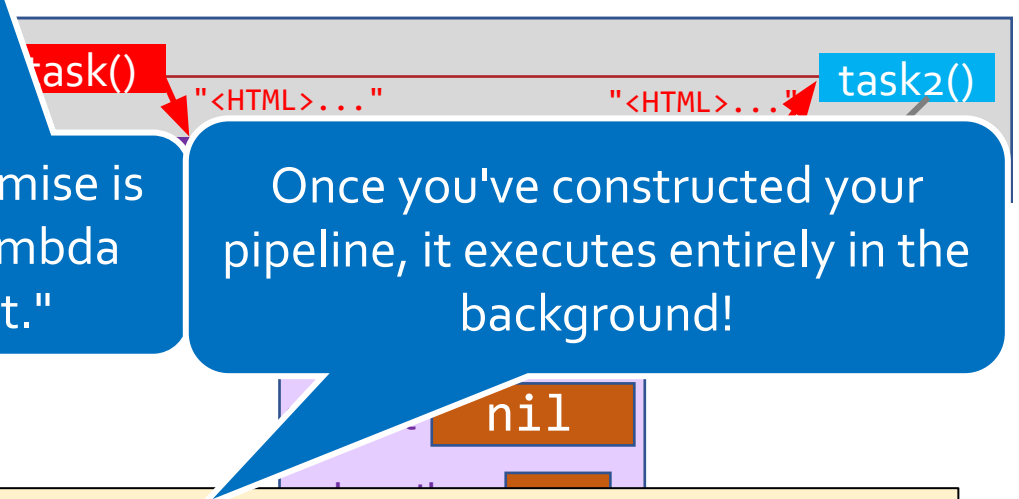


To add a subscriber, you register a lambda function with the Promise object, e.g., p.then(...).

```
p = Promise(...);
p.then(lambda (x)  { task2(x); } );
```

When a Promise is fulfilled, it passes the output/status of the task it was tracking to its subs

This means: "Once the Promise is fulfilled, please call this lambda function with its output."



Once you've constructed your pipeline, it executes entirely in the background!

The subscriber creates a new Promis new task to process the previous stage's output.

```
p = Promise(...);
p.then(lambda (x)  { task2(x); } )
  .then(lambda (y)  { task3(y); } )
  .then(...)
```

You can chain together many subscribers, creating a pipeline of asynchronous routines to process data.

Callbacks Can Get Ugly (exp)

```
function download_in_background_then_run_cb(url, callback1) {
  run_in_background {
    data = connect_and_download_data(url)
    callback1(data)
  }
}

function filt                                    (data, callback2) {
  run_in_b
    fdata
    callba
  }
}

function reorde                                    callback3) {
  run_in_backgr
    sdata = sor
    callback3(sdata);
  }
}

function display_in_ui(sdata) { ... }
```

Our first function

Once it's done we ask it to run a callback to initiate the next stage of pro sing.

ge might
ed data.

Once it's done we ask it to launch a callback to do the next stage of processing.

That final stage might display the data in the UI.

Once i

last stage of processing.

Often, we need to chain a bunch of slow background operations together with multiple callbacks, e.g.:

download a file,
then filter the data,
then sort the remaining data,
then update the UI.

For example, consider the following functions…

To do chain functions like this, people use lambdas.

Let's see!

Callbacks Can Get Ugly (exp)

Let's see how we do end-to-end processing and display our data.

```
function display_sorted_filtered_data(url) {
  download_in_background_then_run_cb(
    url,
    lambda (data) { /* runs filter step in backgr
```

Let's see the details of this lambda function.

Let's see the details of this lambda function.

The first thing we do is call our download function

The lambda calls the

The lambda will be called with the downloaded data once we finish downloading the file.

etails of this nction.

It sorts the filtered data.

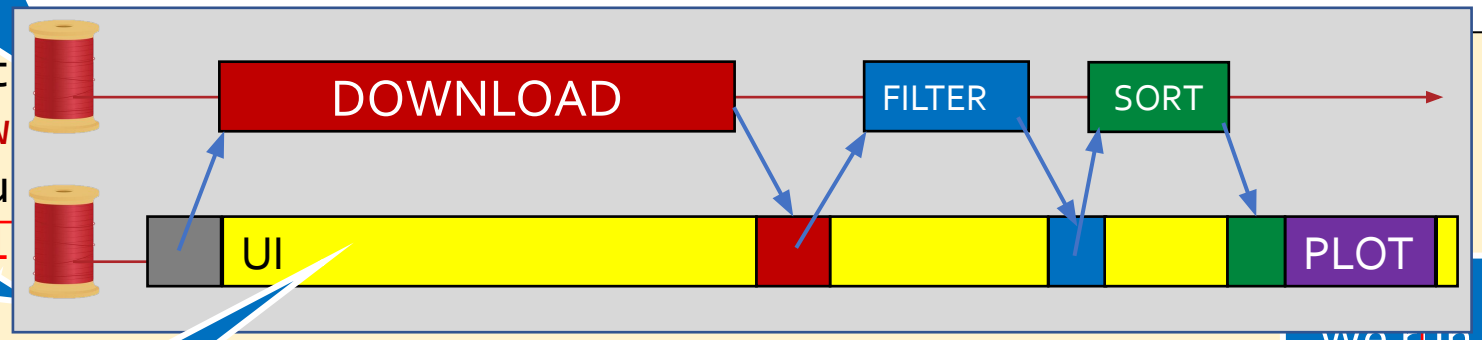
For our callback, we want to pass a lambda that launches the sorting step in the background.

For our callback, we pass in a lambda that performs the final plotting step.

Callbacks Can Get Ugly (exp) - TODO

In this way, we break what would be one long task into a series of shorter background tasks.

When a task finishes, it runs its callback to launch the next background task.



Once we finish downloading the data...

Once we finish filtering the data...

We run a callback that sorts the data...

We run a callback that filters the data...

We run a callback that displays the data...

```
funct
  dow
    u
    l
      data,
      lambda (fdata) {
        reorder_data_in_background_then_run_cb(
          fdata,
          lambda (sdata) {
            display_in_ui(sdata);
          }
        );
      }
    );
  }
);
}
```

Solution #3:Async/Await

```
function download_data(url){
 while (!done) {
   get_data()
   save_data()
 }
 return result;
}
```

```
function filter_data(data) {
 while (!done) {
   foo();
   bar();
 }
 return result;
}
```

```
function sort_data(data) {
 while (!done) {
   divide();
   conquer();
 }
 return result;
}
```

Of course, there's no reason you have to launch tasks serially like our previous example.

Let's see.

Launch the Async Function

You request execution of one or more functions in the background and obtain handle(s) to the pending task(s).

Wait for Completion

You may optionally wait for your task(s) to complete their execution, and then obtain their result.

```
async function run(data1, data2, data3) {
  h1 = run_async task1(data1);
  h2 = run_async task2(data2);
  h3 = run_async task3(data3);

  ... // Do other stuff in the meanwhile...

  result1 = await h1  # block until task1 is done
  result2 = await h2  # block until task2 is done
  result3 = await h3  # block until task3 is done
  use_results(result1, result2, result3)
}
```

Callbacks Can Get Ugly

```
function download_data(url, callback1) {
  run_in_background {
    data = connect_and_download_data(url)
    callback1(data)
  }
}

function filt
  run_in_b
    fdata
    callba
  }
}

function reorde
  run_in_backgr
    sdata = sor
    callback3(sdata);
  }
}

function display_in_ui(sdata) { ... }
```

Our first function

Once it's done we ask it to run a callback to initiate the next stage of processing.

Once it's done we ask it to launch a callback to do the next stage of processing.

That final stage might display the data in the UI.

last stage of processing.

Often we need to chain a bunch of slow background operations together with multiple callbacks, e.g.:

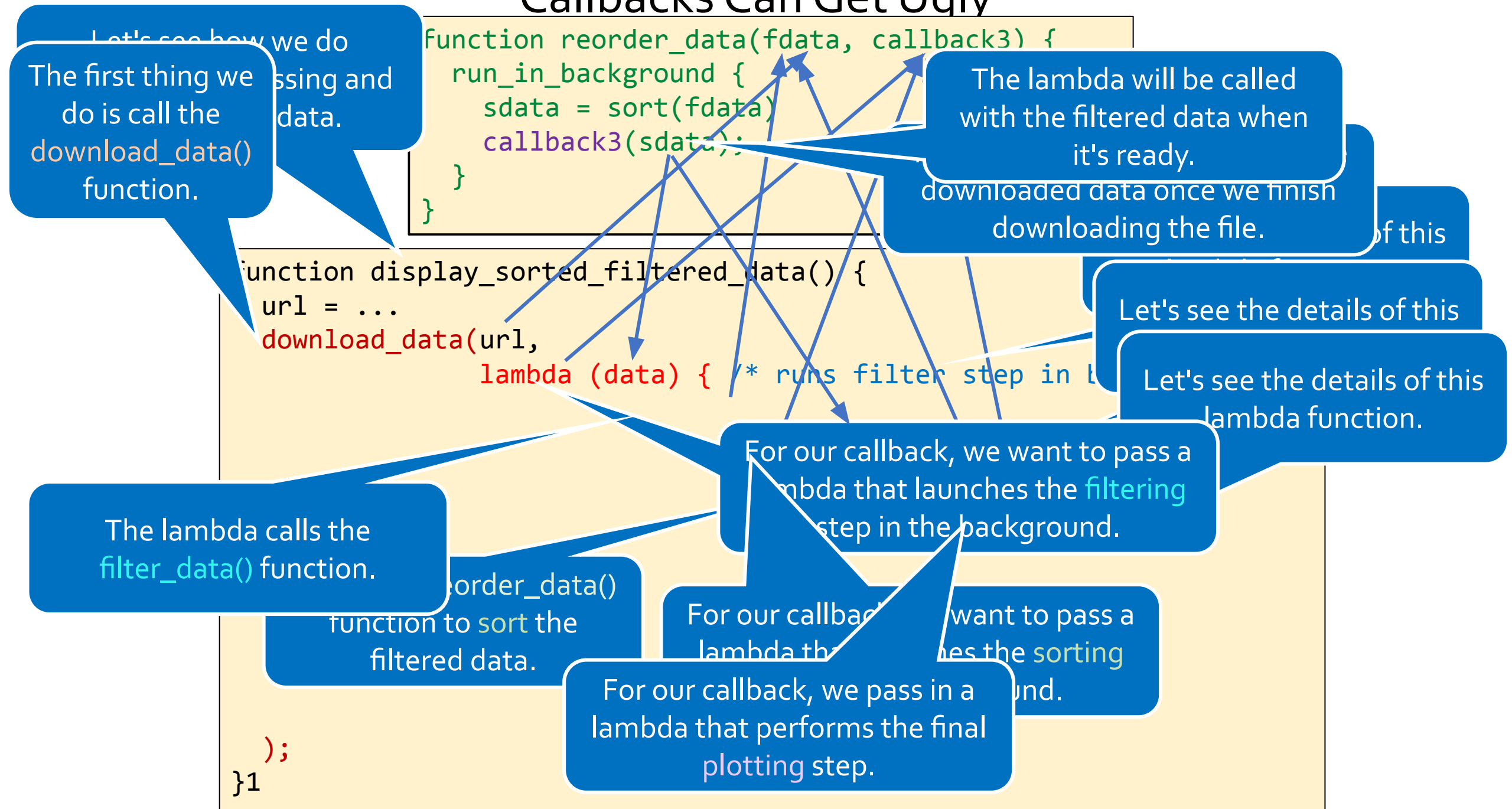
download a file,
then filter the data,
then sort the remaining data,
then update the UI.

For example, consider the following functions...

To do chain functions like this, people use lambdas.

Let's see!

Callbacks Can Get Ugly



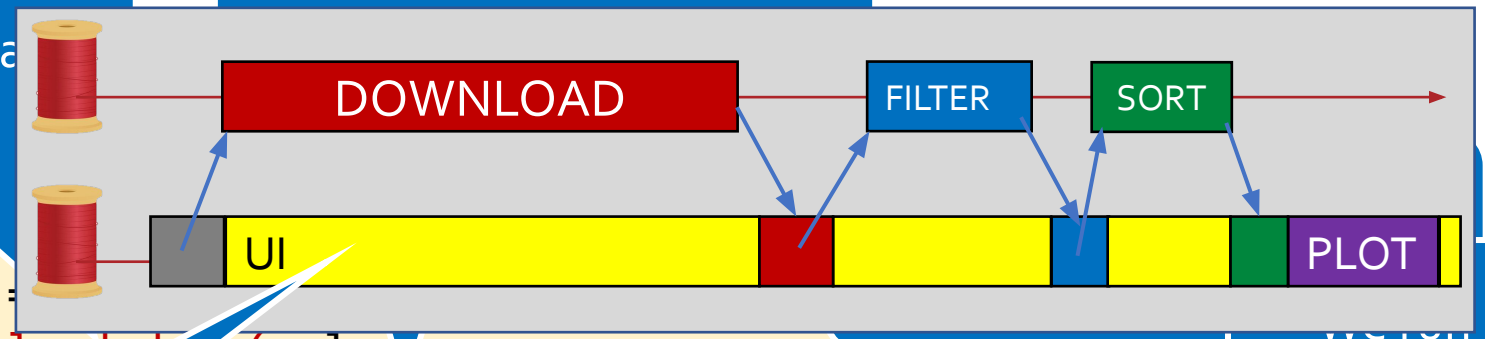
Callbacks Can Get Ugly

In this way, we break what would be one long task into a series of shorter background tasks.

When a task fir nch the next background task.

Once we finish downloading the data

We run a callback that



Once we

a callback that displays the data...

This enables the UI to keep running while we're doing processing.

Once we finish orting the data...

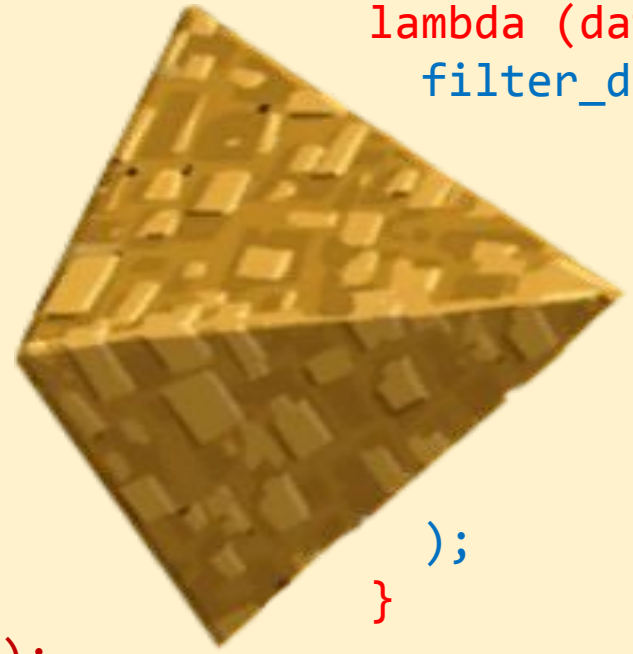
```
funct
  url =
  download data(url,
    lambda (data) {
      filter_data(data,
        lambda (fdata) {
          reorder_data(fdata,
                       lambda (sdata) {
                         display_in_ui(sdata);
                       }
          );
        }
      );
    }
  );
}
```

Callbacks Can Get Ugly

Unfortunately, this nesting can get ugly!

In fact, it's so nasty, it has a name: "callback hell" or "pyramid of doom."

So of course, programming language designers came up with a better alternative.



```
function display_sorted_filtered_data() {
  url = ...
  download_data(url,
                lambda (data) {
                  filter_data(data,
                              lambda (fdata) {
                                reorder_data(fdata,
                                             lambda (sdata) {
                                                display_in_ui(sdata);
                                             }
                                );
                              }
                  );
                }
  );
}
```

Solution #2: Promises

In our previ… then filter the data, and then …e data in the UI.

Wouldn't it … an callbacks? How about this?

Note: You can only use this syntax if you modify these functions to use "Promise objects"!

```
function display_sorted_filtered_data(url) {
  download_data(url)

    .then(lambda (downloaded_data) { filter_data(downloaded_data); }  )

    .then(lambda (filtered_data)   { sort_data(filtered_data);     }  )

    .then(lambda …                 …ui(sorted_data);       }  )
}
```

This chunk of code defines a pipeline of operations.

Each phase of the "pipeline" does its work (in the background), produces a result, and the framework passes that result to the next lambda in the pipeline.

This uses an approach called "Promises."

tion #2: Pr

Whe s, it doesn't run sy ently!

This chunk of code defines a pipeline of operations.

Execution of the pipeline happens concurrently!

```
function display_sorted_filtered_data(url) {
  download_data(url)
    .then(lambda (downloaded_data) { filter_data(downloaded_data); } )
    .then(lambda (filtered_data)   { sort_data(filtered_data);     } )
    .then(lambda (sorted_data)     { update_ui(sorted_data);       } )
}

do_first_thing()
display_sorted_f
while (!done) {
  do_other_stuff();
  do_even_more_stuff();
}
```

on

This pipeline won't run to completion here!
It's just started here, and runs in the background.

```
function download_data(url){
 while (!done) {
   get_data()
   save_data()
 }
 return result;
}
```

```
function filter_data(data) {
 while (!done) {
   foo();
   bar();
 }
 return result;
}
```

```
function sort_data(data) {
 while (!done) {
   divide();
   conquer();
 }
 return result;
}
```

```
function update_ui(data) {
  ...
}
```

The Promise approach is syntactic sugar for callbacks!

Solution #2: Promises

If this step of the pipeline

If this step of the pipeline generates an error.

OK, what about dealing with errors?

) do so, we use an "exception-like" syntax – but it's not exceptions!

```
function display_sorted_filtered_data(url) {
  download_data(url)
    .then(lambda (downloaded_data) { filter_data(downloaded_data);  } )
    .then(lambda (filtered_data)   { sort_data(filtered_data);      } )
    .then(lambda (sorted_data)     { update_ui(sorted_data);        } )
    .catch(lambda (error) =>       { address_all_errors(error);     } )
}
```

This catch e

This catch error handler runs…

And if you don't catch something in-line, you can also catch errors from any phase of the pipeline at the end.

Although this provides less error-handling granularity, and thus should be avoided.

Promises – Easy to Consume, Hard to Build From Scratch

Many languages have libraries that are built on-top of promises.

For example, if you use JavaScript's fetch() function, it supports the .then() and .catch() syntax automatically!

```
function loadContent () {
  fetch("https://www.foo.com/...")
  .then((result) => {
    if (result.status != 200) {
      throw new Error("Bad Server Response");
    }
    return result.text();
  })
  .then((content) => {
    set_text("field", content);
  })
}
```

If you'd like to learn more about Promises, see Timothy Gu's awesome writeup in the slide notes below.

But if you have to build functions from scratch to support Promises, things get much more confusing.

Promises In Different Languages

```
function loadContent () {
  fetch("https://www.foo.com/...")
  .then((result) => {
    if (result.status != 200) {
      throw new Error("Bad Server Response");
    }
    return result.text();
  })
  .then((content) => {
    set_text("field", content);
  })
}
```

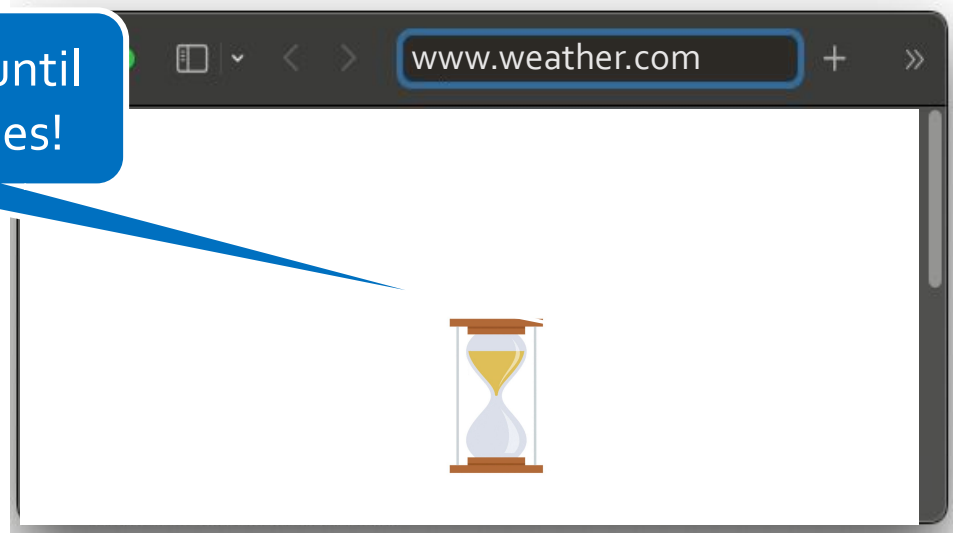
TODO Another language

If you'd like a really great writeup about how JavaScript Promises work, see the slide notes for a writeup by Timothy Gu!

UI Programming Without Concurrency

Here's an example of how java
doesn't use concurrency wo

The browser hangs until the operation finishes!



Clearly, if we have to run a function that takes a while to run, it'll freeze our UI.

So we have to figure out a way to run such a function in the background.

This is called a "blocking" operation.

And then get the result once it's done.

This way, the rest of the page can continue handling user interaction without stalling.

```
<html><body>
<div>The current weather is:</div>
<div id="weather">No weather data</div>

<script>
data = download("www.weather.com/...");
update_html("weather", data);
// other javascript code...
// ...
</script>
</body></html>
```

Callbacks

Idea: You run a long-running task in the background, and give it call a "callback" function to call when it finishes; the callback updates the UI with the task's results.

```
# pseudocode showing use of callbacks

def call_when_done(result):
  # does something with the result
  update_webpage_with_results(resu

def task(callback_fn):
  result = some_long_running_op()
  callback_fn(result)

run_in_background task(call_when_done)

while page_is_not_closed():
  handle_user_interactions_with_page()
```

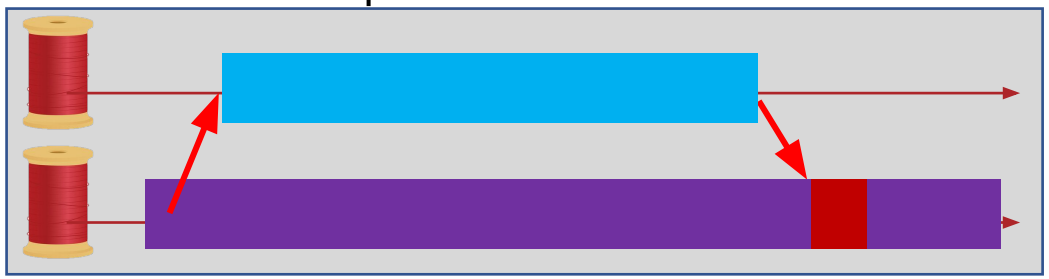
This would correspond to downloading the file in the previous slide.

The callback function accepts a parameter which is the result of a task – it uses the result to update the UI.

function performs an operation, esult, and then calls the callback function with the result.

You launch your task so it runs concurrently with the main thread.

Now your main program can continue running and get notified when the task is complete via the callback.



Example: Handling Button Clicks

```
<html>
<body>
<h1>Button-click Callback Example</h1>

<p>This is what I think:</p>
<p id="truth">USC is great</p>
<button onclick="callbk();">Click Me</button>

<script>
function callbk() {
    document.getElementById("truth").innerHTML +=
        "ly overrated";
}
</script>

</body>
</html>
```

The programmer defines a UI element like a button...

And tells the button what callback function to call...

When an action occurs, like the button is clicked.

llback functions are also used to handle user-driven events in the browser.

This includes the user clicking buttons, dragging items, etc.

Stringing Background Ops Together (hide?)

```
function download_data(url, callback1) {
  run_in_background {
    data = connect_and_download_data(url)
    add_to_run_queue callback1(data)
  }
}

function filter_data(data, callback2) {
  run_in_background {
    fdata = filter_data(data)
    add_to_run_queue callback2(fdata);
  }
}

function sort_data(fdata, callback3) {
  run_in_background {
    sdata = sort(fdata)
    add_to_run_queue callback3(sdata);
  }
}

function display_in_ui(sdata) { ... }
```

Here's pseudocode for what's happening.

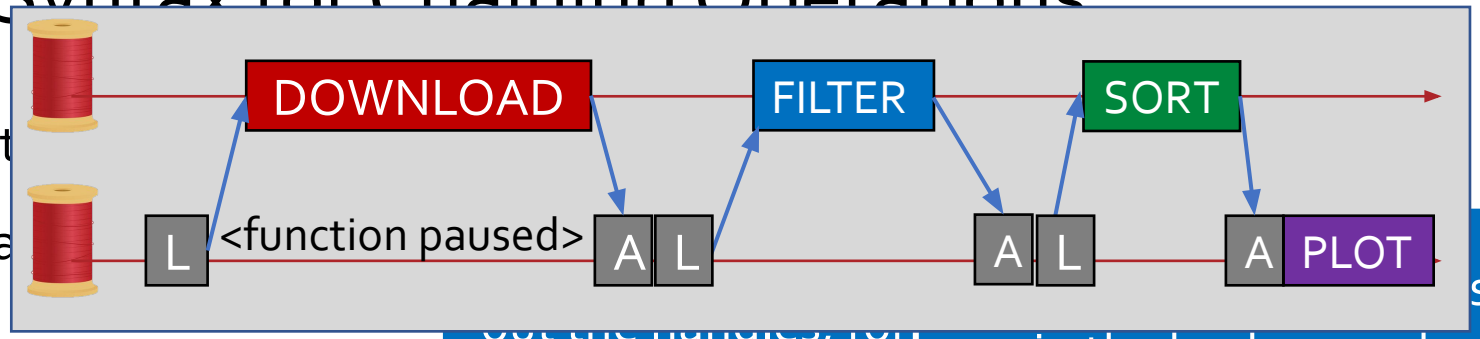
To accomplish this with "real" code,
the syntax is actually much uglier.

Let's see!

Async/Await Syntax for Chaining Operations

The Async/Await model is yet anot

When you use this pa



s)

in the background.

even simpler syntax.

Launch the Async Function

You request execution of one or more functions in the background and obtain handle(s) to the pending task(s).

Wait for Completion

You may optionally wait for your task(s) to complete their execution, and then obtain their result.

```
function display_sorted_filtered_data(url) {
  data = await launch_async download_data(url);
  fdata = await launch_async filter_data(data);
  sdata = await launch_async sort_data(fdata);
  update_ui(sdata);
}
```

After launch, the current function continues running

This provides the same semantics as our promise-based code!

Once we get a result, the thread continues executing.

The code looks synchronous, but in fact it's asynchronous!

Async/Await Syntax for Chaining Operations

Most languages also allow you to wait on multiple handles at the same time before proceeding, almost like a fork-join.

Launch the Async Function

You request execution of a function in the background and obtain a handle to the pending task.

Wait for Completion

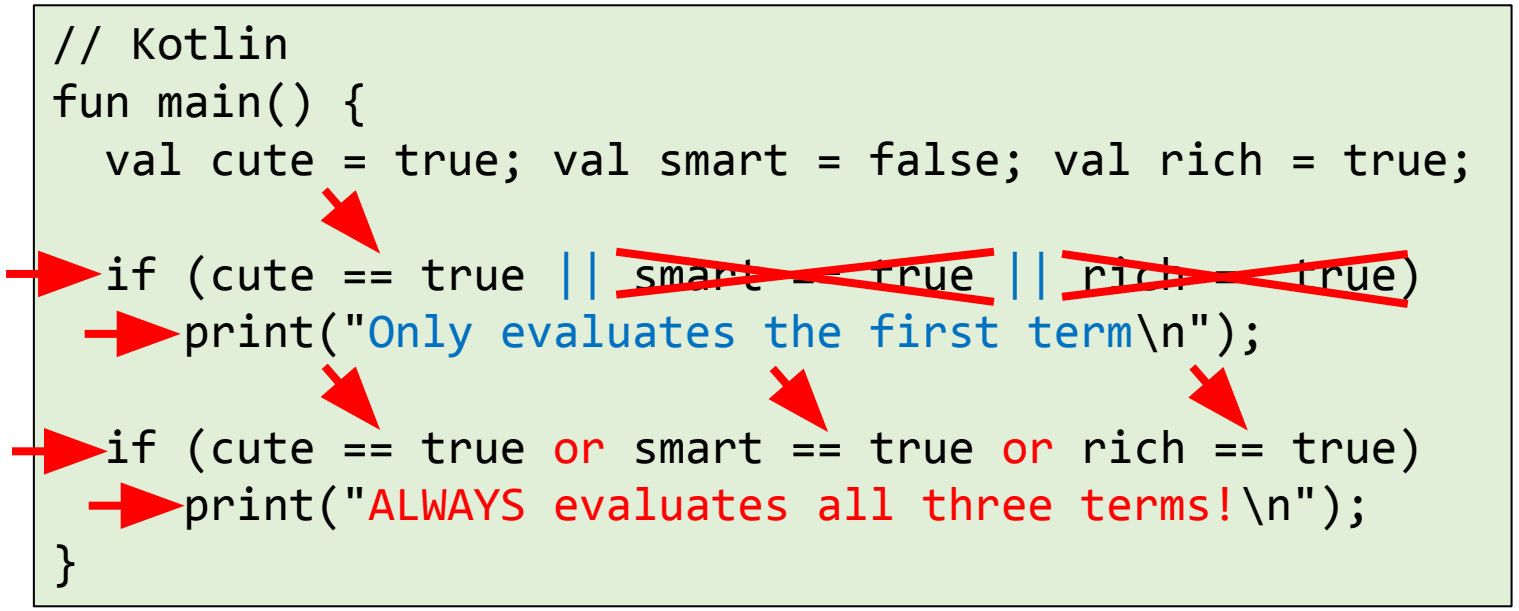
You may optionally wait for your launched task to complete its execution, and then obtain its result.

```
async function run(data1, data2, data3) {
  h1 = launch_async task1(data1);
  h2 = launch_async task2(data2);
  h3 = launch_async task3(data3);

  ... // Do other stuff in the meanwhile...

  await_all [h1, h2, h3];

  ... // Continue here once task1-task3 finish
}
```

Short Circuiting

Virtually all modern languages have some form of short-circuiting.

And some even provide syntax for both options so you can choose to use it or not.



Models for Concurrency

There are two primary approaches for implementing concurrency:

Threading Model

Your program creates multiple "threads" of execution that run concurrently (potentially in parallel).

The programmer explicitly "launches" one or more functions in their own threads, and the OS schedules their execution across available CPU cores.

```
// Multi-threaded program (pseudocode)
void handle_user_purchase(User u, Item i) {
  if (bill_credit_card(u) == SUCCESS) {
    create_thread(schedule_shipping(u, i));
    create_thread(send_confirm_email(u, i));
  }
}
```

Event Driven Model

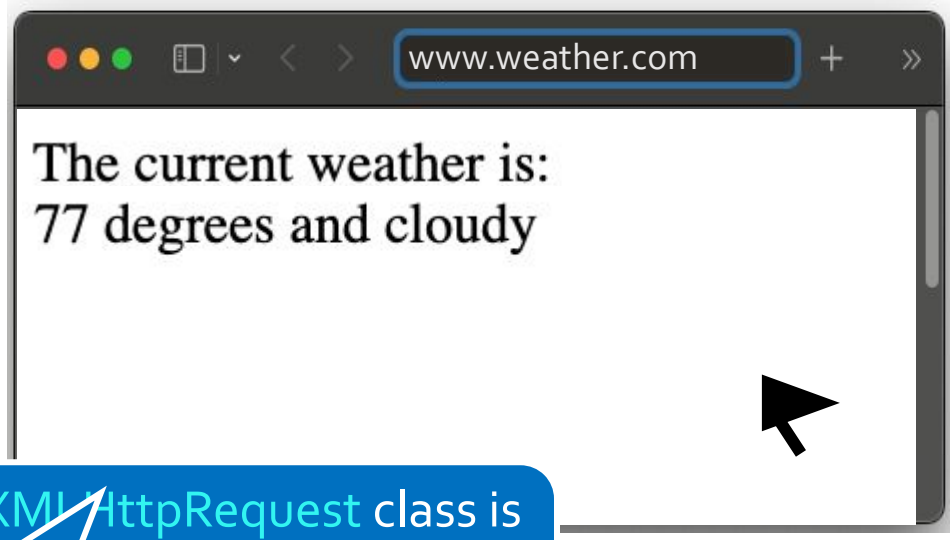
A program is a list of associations, each between an event (e.g., the user clicked a UI button) and a function f() that must be called when that event occurs.

After initial setup, functions are run if/when a related event occurs that triggers the function's execution.

```
// Event-driven program (pseudocode)
function process_payment() { ... }

void setup_event_associations() {
 button = create_new_button("Pay Now!");
 button.set_func(ON_CLICK, process_payment);
}
```

Event Loops, Callbacks and I/O: Example

```
<html><body>
<div>The current weather is:</div>
<div id="weather">No weather data</div>

<script>
function update_weather_text(txt) { ... }

function update_ui_when_data_arrives()
{ update_weather_text(this.responseText); }

function download_weather() {
    var request = new XMLHttpRequest();
    request.onload = update_ui_when_data_arrives;
    request.open("GET", "http://www.weather.com/");
    request.send(); // execution in background
}

download_weather()
update_weather_text("Retrieving weather...")
</script>
</body></html>
```



That callback can update the page's text with the latest weather.

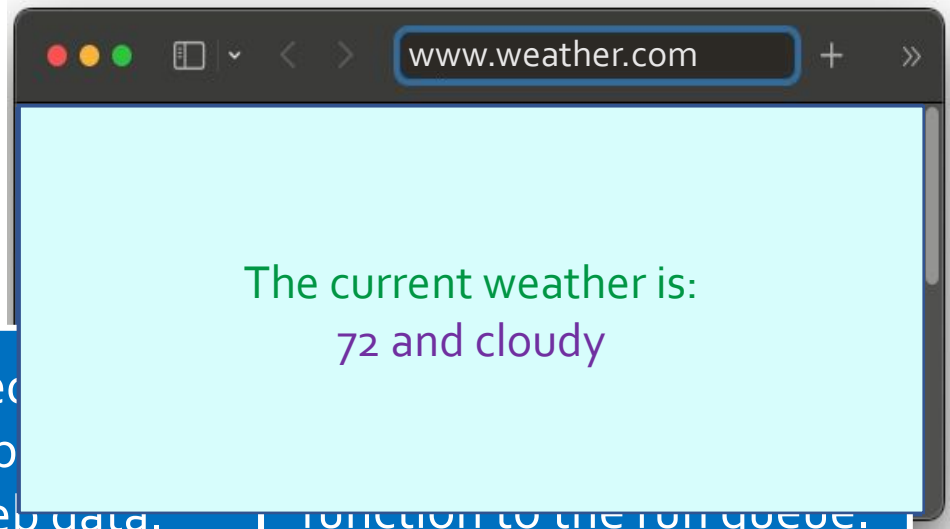
This XMLHttpRequest class is used to download internet data concurrently.

Meanwhile, the user can interact with the page – clicking buttons, etc.

We must specify a callback function to call once the data has been downloaded.

Event Loops, Callbacks and I/O: Example

```
<html><body>
<div>The current weather is:</div>
<div id="weather">No weather data</div>

<script>

function update_weather_in_ui
  /* code to update the weat

function download_weather() {
  var req = new XMLH
  req.onreadystatech
  req.open("GET", "h
  req.send(); // sta
}

download_wea
set_backgrour

</script>
</body></html>
```



A few seconds later, our web request finishes and the event triggers.

We then dequeue the next function call and run it to completion.

Our function continues running immediately…

XMLHttpRequest
on_completion = update_weather_in_ui()

run_queue

```
update_weather_in_ui();
set_background_color(BLUE);
```

Event Loops and Sequences of Callbacks

```
<html><body>
<div>The current weather is:</div>
<div id="weather">No weather data</div>

<script>

function update_weather_in_ui()
 { /* code to update the weather field */ }

function download_weather() {
  var req = new XMLHttpRequest();
  req.onreadystatechange = update_weather_in_ui;
  req.open("GET", "https://www.weather.com/...", true);
  req.send();
}

download_weather();
set_background_color(BLUE);

</script>
</body></html>
```

In this example, we saw how to run a long-running op in the background…

And use a callback upon completion to perform the next processing step.

While sometimes we'll just need to run a single background op…

In many cases, we want to chain multiple long-running background ops together into a pipeline.

This lets us accomplish more complex tasks in the background.